UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Ingeniería en Informática

Proyecto Fin de Carrera

"Creación de un manual para el diseño e implementación de videojuegos en *Blender* 2.49b"

Autor: Roberto Peral Castro

Tutor: Juan Peralta Donate

20 Marzo, 2010

Agradecimientos:

- A mi familia, que siempre me ha apoyado en todos los aspectos de mi vida, y especialmente en los estudios.
- A mi tutor D. Juan Peralta Donate, por haberme dado la oportunidad de realizar este proyecto, y por el entusiasmo mostrado a lo largo de todo su desarrollo.
- A mis compañeros de beca, por su comprensión.
- A mi pareja, que ha demostrado ser una gran amiga y compañera, y cuya ayuda a lo largo de toda la carrera me ha permitido llegar hasta aquí con menos contratiempos de los que podrían haber ocurrido debido a mi carácter despreocupado y despistado.

Contenido

1. Introducción

Hoy en día, el éxito de las empresas de videojuegos, tanto en España como en el resto del mundo, es una realidad. Importantes empresas relacionadas con el mundo del videojuego, tales como Microsoft Games [1], Ubisoft [2] o Electronic Arts [3], además de otras muchas que están empezando a surgir o que ya están consolidadas, tienen oficinas en España, y pueden suponer perfectamente una fuente de empleo para futuros ingenieros informáticos.

Los campos de la informática que involucra el diseño de videojuegos son entre otros: bases de datos, inteligencia artificial, redes y diseño 3D. En definitiva, y dependiendo del juego que se quiera diseñar, el desarrollo puede abarcar todas las disciplinas de la informática, e incluso otras totalmente distintas como arte, música, historia o estrategia.

Un proyecto de videojuego puede ser un proyecto muy amplio y complicado que involucra a un gran equipo de profesionales. Por esta razón un ingeniero informático (que conoce la mayoría de las disciplinas de la informática en cierta medida, y que ha aprendido a gestionar proyectos y trabajar en equipo), sería la persona ideal para dirigir o trabajar en un proyecto de este estilo.

El hecho de que la profesión de desarrollador de videojuegos, sea una profesión factible hoy en día, y la pasión personal por el mundo del videojuego, ha dado como resultado la decisión de iniciar este proyecto, en el que el objetivo principal era la creación de un videojuego.

En un principio este proyecto iba a ser la implementación de un videojuego, en un motor de juego concreto. Esta idea cambió a medida que se fue avanzando en el proyecto, ya que se observó que el trabajo a realizar era muy amplio, y los problemas y errores encontrados, muy numerosos.

Todo el conocimiento obtenido al resolver esos problemas se estaba perdiendo, ya que no se vería reflejado en la memoria ni en el código. Se pensó entonces, en escribir un manual que comentase como subsanar esos problemas de antemano, que definiese las capacidades del motor de juego, y como desarrollarlas al máximo. Con un manual de este estilo, se mejoraría tanto la calidad del diseño como el tiempo de desarrollo.

Mirando estas ideas desde un punto de vista práctico, se decidió que un manual sobre el uso de *Blender* [4] para la creación de un juego, sería de mayor utilidad para futuros proyectos realizados con dicha herramienta. También se pensó, que como en cualquier asignatura, utilizar el videojuego implementado para proporcionar ejemplos sería muy útil para hacer entender mejor el funcionamiento del programa.

1.1.Descripción de los capítulos del proyecto

1- *Introducción*

En este apartado se hace una breve introducción sobre el estado actual de la industria del videojuego y el papel que puede desempeñar un ingeniero informático en ella. Explica además el tema principal sobre el que trata el PFC y las motivaciones que llevaron al autor a realizarlo.

2- *Objetivos del proyecto*

Los diferentes objetivos del que se pretenden alcanzar en este PFC, se ven enumerados en este apartado, y además se aclara lo que no se pretende con este proyecto. Servirá entonces para que el lector se haga una idea de lo que se va a encontrar a medida que avance en la lectura de esta memoria.

3- *Estado de la cuestión*

El estado de la cuestión aborda el tema de los motores de juego, pasando por su pasado, presente, y futuro próximo, dando así forma al contexto que envuelve la definición de un motor de juego. Se revisará la evolución de estos motores, a lo largo de la historia, explicando las características aportadas por cada uno de ellos a medida que se avanza en la línea temporal. También se explicará su estado actual, aportando un ranking de los mejores motores del mercado del 2009, y explicando las posibles características a tener en cuenta a la hora de medir la calidad de un motor de juego.

En un sub-apartado Se llevará a cabo una descripción de las características y capacidades de *Blender*, se introducirá brevemente el proyecto "*YoFrankie*", un videojuego creado íntegramente en *Blender*, y por último se enumerarán las razones por las que se ha seleccionado *Blender* para realizar el manual.

4- *Manual de desarrollo de videojuegos en Blender*

El manual es el objetivo final del PFC en sí mismo, por ello no es de extrañar que este apartado sea el más extenso de toda la memoria y el que explica todo lo necesario para aprender a programar videojuegos en *Blender*. Primero se describe una pequeña guía con los pasos básicos ordenados, necesarios para crear un videojuego. Seguidamente se hace una breve introducción a los conceptos y ventanas básicos del interfaz como una primera toma de contacto del lector con *Blender*. Y por último, se explican ampliamente, las técnicas y capacidades utilizadas en *Blender* para crear videojuegos. Se explican temas como la animación de personajes, el motor de físicas, inserción de lógica en los objetos del juego, creación de efectos especiales, y todo esto aplicado a la interfaz grafica de *Blender*.

5- *Conclusiones*

Las ventajas y desventajas que se han obtenido tras el aprendizaje de uso de la herramienta *Blender* vienen reflejadas en este apartado, en el que además se exponen las conclusiones personales que han surgido tras la realización del proyecto.

6- *Líneas Futuras*

En este apartado se expondrán posibles ampliaciones del proyecto, o usos prácticos que se puedan dar a este manual en un futuro.

7- *Gestión del proyecto*

Puesto que este PFC es un manual, sólo se expondrá la planificación temporal del proyecto mediante un diagrama de Gantt, y se comentará la diferencia entre el tiempo planificado y el tiempo real que ha llevado la realización del mismo.

8- *Bibliografía*

Referencias usadas para el desarrollo del proyecto.

1.2.Acrónimos y definiciones

ACRÒNIMO	DEFINICIÓN
PS2	Play Station 2, consola de videojuegos de la compañía SONY.
API	Application Programming Interface - Interfaz de Programación de Aplicaciones. Grupo de rutinas que provee un sistema operativo, una aplicación o una biblioteca, que definen cómo invocar desde un programa un servicio que éstos prestan.
DLL	Dynamic Linking Library - Librería de Enlace Dinámico. Es la implementación de Microsoft del concepto de bibliotecas (librerías) compartidas en sistemas Windows y OS/2.
OpenGL	Open Graphics Library. Conjunto de especificaciones estándar que definen una API multilenguaje y multiplataforma para escribir aplicaciones o juegos que producen gráficos en 3D.
Antialiasing	Cualquier técnica que reduce la apariencia accidentada de los bordes en imágenes digitales tanto planas como en tres dimensiones
Bump mapping	Técnica de gráficos computacionales 3D que consiste en dar un aspecto rugoso a las superficies de los objetos.
Rigging	Acción de asignar los vértices de una malla a los huesos de un esqueleto..
Cuaternión	Los cuaterniones son una extensión de los números reales generada añadiendo las unidades imaginarias: i, j y k a los números reales y tal que $i^2 = j^2 = k^2 = ijk = -1$.
Shader	Es una tecnología reciente y que ha experimentado una gran evolución destinada a proporcionar al programador una interacción con la GPU hasta ahora imposible. Los shaders son utilizados para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla.
Open source	Código abierto. Software distribuido y desarrollado libremente.
Royalty free	Licencia de uso en la que se otorga la cesión de derechos de reproducción no exclusivos, a no ser que se pacte lo contrario; de carácter perpetuo o permanente no sublicenciable o transferible, con la excepción del cliente final.
3dfx	Compañía fabricante de placas aceleradoras de gráficos.
Renderizar	Generar una imagen (imagen en 3D o una animación en 3D) a partir de un modelo, usando una aplicación de computadora.
Streaming	El "*streaming*" hace posible reproducir contenidos multimedia sin necesidad de ser descargados previamente, almacenándolos en un buffer.
Sprite	Tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado sin cálculos adicionales de la CPU.
Voxel	Unidad cúbica que compone un objeto tridimensional.
Frame	Imagen particular dentro de una sucesión de imágenes que componen una animación.
Stickman	Personaje dibujado con una esfera por cabeza y 5 palos como torso y extremidades.

Tabla 1. Acrónimos y definiciones

2. Objetivos del proyecto

El objetivo principal de este proyecto, es crear un manual de referencia rápida, que enseñe a diseñar un videojuego del modo más sencillo y guiado posible, con la herramienta *Blender* en su versión 2.49b. No se pretende que sea un manual teórico que explique las propiedades físicas de los objetos, los distintos tipos de texturas, o los métodos de programación aplicables a los videojuegos.

Con las herramientas de diseño que existen hoy en día, y haciendo una medición basada en la experiencia de haber realizado este proyecto, el trabajo necesario para implementar un juego, es muy superior a la dificultad conceptual que implica en sí misma esta implementación.

Se ha estimado que la relación entre esfuerzo y dificultad a la hora de hacer un videojuego es de un 80% de esfuerzo frente a un 20% de dificultad. De ese esfuerzo, la mitad se debe al aprendizaje necesario para poder utilizar la herramienta de diseño.

Con este manual, se pretende reducir más de la mitad, el esfuerzo dedicado al aprendizaje de uso de la herramienta. De este modo, se consigue que alguien que quiera realizar un videojuego en *Blender*, no pierda tiempo investigando todas las opciones que ofrece el motor de juego. Además permite al diseñador centrarse más, en cómo quiere que luzcan y se muevan sus personajes o qué historias y eventos quiere crear para el juego. Los porcentajes estimados, puesto que la dificultad no va a incrementarse, quedarían del siguiente modo: 50% trabajo, 20% dificultad, y un 30% que podría dedicarse a otras fases de un proyecto, como la planificación o el diseño.

Los juegos son los programas que más disciplinas de la informática pueden llegar a usar. Por ejemplo, bases de datos, diseño 3D, Redes, Interfaces, Inteligencia artificial; Y toda esta cantidad de conocimiento puede resultar abrumadora a la hora de ponerlo en práctica, si no se tiene una cierta organización.

Es fácil que una persona se desmoralice y abandone un proyecto, cuando ve una cantidad de trabajo ingente por delante, y no tiene focalizado el punto por el que empezar. Así que en resumen, los objetivos principales de este manual son:

Que el lector, diseñador, programador, ingeniero, da igual el tipo de persona, sea capaz de crear un videojuego en *Blender* de un modo rápido y ordenado.

Lograr que se obtengan resultados en pocos días, aumentando así considerablemente los ánimos y las ganas de continuar con el proyecto de videojuego.

Incentivar la creación de futuros proyectos con la herramienta *Blender*.

Para conseguir realizar este manual, se ha creado un juego de lucha, sólo con un personaje y un único escenario. El hecho de sólo tener este reducido repertorio es a causa de que el objetivo de realizar dicho videojuego, no es el juego en sí mismo, si no obtener la mayor cantidad de conocimientos acerca del funcionamiento de *Blender* para el diseño de videojuegos. Habría sido por tanto, contraproducente diseñar más personajes y escenarios, cuya implementación, supondría una enorme cantidad de trabajo que no aportaría nuevos conocimientos al proyecto.

3. Estado de la cuestión

3.1.Descripción de un Motor de juego o Game Engine

Un motor de juego es un sistema software diseñado para crear y desarrollar videojuegos. Unos motores están diseñados para crear juegos para videoconsolas, y otros están diseñados para crear juegos para PC. Algunos de ellos son multiplataforma, y permiten diseñar para varias videoconsolas y para PC.

Las funcionalidades típicas proporcionadas por un motor de juego son:

- Un motor de renderizado, 2D o 3D, para dibujar por pantalla los gráficos del juego.
- Un motor de físicas y detección de colisiones, para simular propiedades como la gravedad y detectar choques entre objetos, contacto con el suelo, simulación de fuerzas, etc.
- Soporte para sonido, tanto edición como integración en el juego.
- Soporte para animación, normalmente se trata de animación esqueletal.
- Soporte para scripting.
- Soporte para redes.
- Inteligencia artificial.
- "*Streaming*".
- Gestión de memoria.
- Soporte multihilo.

Los motores de juego proporcionan un entorno de trabajo que integra todas las características anteriores normalmente con un interfaz visual, que permite un modo más fácil de diseñar y programar un videojuego. Estos motores abstraen la lógica de los videojuegos, hasta el punto que se puedan hacer varios juegos distintos con el mismo motor, reduciendo así costes, complejidad, y tiempo de desarrollo, factores todos ellos críticos en la industria del videojuego.

Los motores normalmente son independientes de la plataforma, eso es lo que pasa con Ogre [5], o *Blender* por ejemplo, que puede compilar los juegos para Windows, Linux y Mac además de otros sistemas operativos. Algunos de estos motores de juego están implementados de una forma modular para que sea más fácil cambiar un motor de físicas por otro, un gestor de sonidos por otro, y en definitiva cualquiera de los módulos que forman el motor de juego.

Algunos motores de juego, incluyen herramientas de diseño 3D, para integrar más fácilmente los modelos 3D, con los elementos que implementan la lógica para dichos modelos. Hay otros que no disponen de un software para crear sus propios modelos, pero que dan soporte para varios formatos de programas de diseño 3D.

En los apartados sucesivos se hablará del origen de los motores de juego, los tipos que hay, de motores de juego concretos, y de *Blender*, el motor que se eligió para desarrollar este proyecto.

3.2.Historia de los motores de juego

Al principio, crear un juego, significaba desarrollar y compilar el código desde cero, método indiscutiblemente muy costoso y poco eficiente. El verdadero cambio llego con el juego "*Doom*". Pese a que ya existían motores de juego anteriores al del famoso "*Doom*" de la compañía "*id software"* [6], fue este el que supuso una verdadera revolución en la reutilización de código.

A continuación se expondrá la evolución de distintos motores de juego, desde antes del "*Doom*", hasta los motores más actuales. En este listado no están todos los motores de juego, ya que son muchísimos, pero nos dará una idea de su evolución.

Uno de los primeros motores existentes, fue "*Space Rouge*", que más tarde evolucionó en "*Ultima underworld*". Esta segunda versión incluía un algoritmo para realizar mapeado de texturas, y permitía crear paredes con distintas alturas, y planos inclinados para dar un aspecto 3D al entorno. Mientras que los personajes eran "*sprites",* o dibujos planos en 2D, el mundo por el que se movían se renderizaba en 3D. Salió al mercado en 1990.

Ilustración 1. *Space Rouge*

Ilustración 2. Ultima underworld

El siguiente motor de juego analizado es el "*idTech*", que se desarrolló a partir del anteriormente mencionado "*Doom*". Este motor no era un motor 3D, si no un motor 2D que representaba tanto objetos y personajes mediante imágenes en 2D en movimiento. A pesar de que "*Doom*", es un juego en 2D, todavía se considera un título 3D, gracias a su capacidad de simular objetos 3D mediante imágenes 2D móviles, que se colocaban en el juego a una altura diferente de la del entorno. "*Doom*" salió al mercado en 1993 y con su motor gráfico se crearon numerosos juegos.

Ilustración 3. Doom

"*Voxel*" hizo su aparición en 1992, y se diferencia de los anteriores en el modo que tenía de representar los objetos y el terreno en 3D. "*Voxel*" conseguía una representación en 3D mediante bloques 2D de diferentes alturas. Un "*Voxel"* es en un objeto 3D, lo que un pixel es en una imagen en 2D. Los "*voxels*" no contienen su posición definida por coordenadas, si no que esta se deduce de su posición en el fichero de datos, a diferencia de los datos vectoriales, que tienen sus propias coordenadas.

Este motor aporta una manera distinta de representar gráficos en 3D, de modo que fuesen más detallados y con unos contornos más suaves. "*Voxel*" dio lugar a algunos juegos muy conocidos como: *Blade Runner* (para personajes y artefactos), *Comanche*, *Command & Conquer: Tiberian Sun and Command & Conquer: Red Alert 2* (para la mayoría de vehículos), *Delta Force*, *Master of Orion III* (para las naves y sistemas solares).

"*Build*" [7] es el motor de juego sobre el que se construyó, "*Duke Nukem 3D*" [8]. Es un motor muy similar al del "*Doom*", en el sentido de que crea un mundo sobre planos 2D, poblado por "*sprites".* El motor "*Build*" permitía mirar arriba y abajo con el ratón, y mediante etiquetas situadas en el suelo o paredes, permitía tele trasportar al jugador a nuevas zonas, simulando que caía por un agujero o atravesaba un portal. Este motor salió al mercado en 1993 y produjo numerosos títulos como: *Blood, Duke Nukem 3D, Extreme Paintbrawl, PowerSlave, Redneck Deer Huntin', Redneck Rampage, Redneck Rampage Rides Again, Shadow Warrior, William Shatner's TekWar, Witchaven, Witchaven II*

Ilustración 4. Duke Nukem 3D

En 1995 aparece "*XnGine*" uno de los primeros motores gráficos en 3D que estaba basado en DOS. Implementado por Bethesda Softworks [9] en un principio este motor tuvo algunos problemas, y los jugadores se quedaban atascados en los objetos 3D del juego. En versiones posteriores se haría compatible con 3dfx, permitiendo la creación de enormes mundos virtuales.

Ilustración 5. XnGine

Muy avanzado para su época, el motor de juego "*Jedi*" hizo su aparición en 1995. Este motor no creaba en realidad objetos 3D durante la ejecución del juego, sino que los desarrolladores creaban un modelo en 3D, obtenían imágenes desde 32 ángulos consecutivos, y el motor hacía el renderizado de dichas imágenes, escalándolas a medida que el jugador se acercaba o alejaba de los objetos. Otra mejora que introdujo este motor, era la posibilidad de saltar y agacharse, y como en el "*XnGine*", se permitía mirar arriba y abajo.

Ilustración 6. Jedi

El primer motor 3D de id Software apareció en 1997 y fue el que usó para diseñar el juego "*Quake*". Este motor tenía varias maneras de evitar tener que procesar enormes cantidades de datos que ralentizasen el juego. Uno de estos métodos era evitar el renderizado de las zonas que el jugador no pudiese ver. Para llevar a cabo esta tarea, las habitaciones estaban compuestas por objetos que creaban habitáculos cerrados, y se utilizaba un preprocesador de renderizado, que identificaba la parte trasera de dichos objetos. Esa parte trasera no se cargaba, al igual que todo el espacio que se encontraba fuera de la habitación ocupada por el jugador.

Esta técnica normalmente reducía el número de polígonos a la mitad o menos. Otra técnica que utilizaba para reducir la demanda de CPU, era el uso de "*z-buffering*" [10], que servía para calcular que objetos o zonas del juego no eran visibles, y no renderizarlas.

El motor del "*Quake*" también añadió puntos de luz, y soportaba aceleración por hardware. Posteriormente se añadiría soporte para "*OpenGL*" [11].

Ilustración 7. Quake

El motor de juego "*Renderware*" [12], tiene más de 200 títulos, la mayoría de ellos para la PS2, pero también algunos para PC, y para muchas de las videoconsolas actuales.

Nacido en 1998 este motor precedió a la aceleración por hardware, y eso pudo ser la causa de que hoy en día no sea un ejemplo a seguir como API para la creación de videojuegos.

Uno de los puntos fuertes de "*Renderware*", era la capacidad de modificar el arte y los procesos del juego en tiempo real. De este modo un desarrollador puede cambiar el color de los objetos, o modificar las físicas de una acción, sin modificar el código que lo implementa, ni hacer el renderizado de toda la escena de nuevo.

Ilustración 8. Renderware ejemplo1

Ilustración 9. Renderware ejemplo 2

"*Quake*" [13] ya proporcionaba aceleración 3D, pero El motor "*Quake 2*", también conocido como "*idtech 2*", proporcionaba soporte OpenGL nativo. Otros aportes que hizo este motor de juego fueron la inclusión de luminosidad de diferentes colores, y un nuevo modelo de juego basado en librerías dinámicas escritas en C, en vez del antiguo modelo basado en Cscripting.

Creado en 1997, y debido al modelo de juego implementado, este motor permitía liberar la parte del código fuente referente al Modding, para que la comunidad de jugadores pudiese modificar el aspecto del juego, manteniendo el resto del código del motor propietario.

"*GoldSRC*" [14] fue uno de los motores que dio un gran impulsó al mundo de los videojuegos en el PC, gracias a su soporte de OpenGl y Direct3D. También dio un buen impulso a la venta de tarjetas gráficas, y dio lugar a juegos míticos como: *Half-Life, Team Frotress, Day of Defeat, and Counter Strike.*

Ilustración 10. GoldSRC

En 1998 apareció el motor "*Unreal*" [15], llamado igual que el juego que lo vio nacer, y pese a ser en un principio, un motor pensado para juegos de disparos en primera persona, también se utilizó para crear algunos juegos de rol online.

"*Unreal*" tenía su propio sistema de scripts, un editor de mapas, y un programa de modificación. Soportaba aceleración por hardware y por software, detección de colisiones, y filtrado de texturas. Debido a estas nuevas capacidades, los juegos creados con este motor, requerían tarjetas gráficas bastante potentes.

Un ejemplo de motor que utilizaba "*voxel*", fue "*Outcast*". Este motor no necesitaba una tarjeta gráfica para dar lo mejor de sí mismo, pero si un buen procesador. Debido a estar basado en aceleración software, era más común hacer juegos de puzles o de aventuras gráficas, que requieren menos velocidad.

El hecho de que este motor estuviese basado en un motor "*voxel*", permitía hacer el renderizado de zonas de paisajes a mucha más distancia de lo que podía hacerlo un motor poligonal. Además, este motor creado en 1999, incluía capacidades como: *bump mapping*, *anti-aliasing*, sombreado dinámico, un avanzado sistema de partículas, animación por esqueletos, y soporte para primera y tercera personas.

Ilustración 11. Unreal

Ilustración 12. Outcast

También en 1999 apareció "*Quake 3*" o "*idTech3*", renovando en gran medida su antiguo motor, sustituyendo la animación a partir de esqueletos, por la animación a partir de vértices. Este método puede dar lugar a animaciones más elaboradas, aunque también consume mucho más tiempo de desarrollo. Esta versión del motor "*Quake*", también introdujo superficies curvas, colores de 32 bits, y capacidades de red avanzadas para su época. Todas estas capacidades requerían una tarjeta gráfica con soporte para OpenGL.

"GeoMod" [16] es un motor de juego, cuya principal característica era la capacidad de modificar el terreno en el que se movían los personajes, por el efecto del impacto de una bala, el choque de un vehículo o cualquier tipo de evento. Fue creado en 2001, pero hora se encuentra en desuso.

"*Gamebryo"* [17] fue diseñado para soportar varias plataformas, escrito en C++, tanto para consolas como para ordenadores. Además está coordinado con los drivers "*PhisX*" de "*Nvidia*", incluye "*DirectX* 9 y 10", soporta desarrollo multihilo, detección dinámica de colisiones, sistemas de partículas y audio 3D. Otra capacidad interesante a la hora de desarrollar un videojuego, que proporciona este motor, es la compatibilidad e integración con editores 3D como "Maya" y "3DS Max".

"*Gamebryo*" vio la luz en el 2003, y se estima que con él, se han creado más de 200 juegos, algunos tan famosos como: *Dark Age of Camelot, Empire Earth II y III, Fallout 3, Oblivion, Prince of Persia 3D, Sid Meier's Civilization IV.*

"*Source*" es el motor sobre el que se diseñó *Half Life 2*, alabado como el mejor juego de la historia. *Valve*, la compañía propietaria del motor sigue retocando el motor hoy en día, lo que propicia que desde su creación en 2004, aun se siga utilizando para la implementación de videojuegos. Este motor se enfrenta al desarrollo de videojuegos por todos los frentes, implementa iluminación y sombreados dinámicos, físicas, efectos realistas tales como superficies acuáticas y *motion Blur*. Respecto al apartado técnico, incluye optimizaciones multiprocesador y una arquitectura de red eficiente. Juegos famosos creados con el motor "*Source*" son: *Counter Strike: Source, Dark Messiah of Might and Magic, Garry's Mod,Half Life 2, Left 4 Dead, Portal, Postal III*

Ilustración 13. *Gamebryo*

Ilustración 14.*Half Life 2*

"*Euphoria*" [18] no es un motor de juego en sí, si no un motor de animación. Se incluye en este apartado debido a que se puede utilizar en otros motores de juego, como ocurre con "*RAGE*", pero sobre todo, por la gran innovación que aporta a la animación en videojuegos. "*Euphoria*" no sólo utiliza un esqueleto unido a la malla que representa al personaje para realizar los movimientos, sino que incluye una simulación de los músculos, el sistema nervioso, y un sistema de colisiones avanzado, para conseguir que los movimientos del personaje sean más realistas y distintos en cada interacción con su entorno. Esto es posible porque las animaciones no se crean previamente, sino que se generan espontáneamente durante la ejecución del juego, dependiendo de las interacciones ocurridas con el entorno, ahorrando así mucho tiempo de desarrollo, y obteniendo resultados más realistas.

Ilustración 15. *Euphoria* esqueleto

Ilustración 16. *Euphoria* músculos

Ilustración 17. *Euphoria* colisiones y físicas

Con esta revisión de algunos de los motores de juego más famosos de la historia, se puede deducir la línea que han ido siguiendo los desarrolladores a lo largo de esa historia. Dejando a un margen el objetivo principal de los motores de juego, (reutilizar código de un modo fácil y eficiente), se podría decir que la sucesión de mejoras de los motores ha sido la siguiente:

"En un principio, parece que los desarrolladores se preocupaban más por el avance gráfico de los juegos. Ponían sus esfuerzos en conseguir un efecto visual de tres dimensiones, que las máquinas de la época fuesen capaces de soportar. Ya fuese con sprites, voxels, o con el 3D real que acabaría llegando más tarde. Cuando la aparición de las tarjetas gráficas se hizo patente, y conseguir buenos gráficos no era tan complicado, se centraron en añadir más capacidades a los juegos. Estas capacidades eran nuevos movimientos para los personajes, más interacción con los escenarios o la vista en tercera persona. La preocupación por los juegos online llegaría con el boom de internet, y en los últimos tiempos, además de mejorar todos los aspectos ya mencionados, se le da mucha importancia al realismo que proporcionan los juegos, centrándose en iluminación y sombreados, detección de colisiones, mapeado de texturas superrealistas, o animación corporal y facial."

Los motores expuestos anteriormente, han sido extraídos de un artículo debido a su colocación cronológica y los detalles sobre la evolución proporcionada, pero existen muchísimos más, tanto propietarios como libres, entre los que está "*Blender Game Engine*", el objeto de este PFC.

Nombre del motor	Fecha	Mejoras aportadas
Space Rogue/ Ultima Underworld	1990	Mapeado de texturas Planos inclinados Sprites.
idTech/Quake	1993	Imágenes 2D móviles para personajes y objetos
Voxel	1992	Representación 3D con *Voxel*s.
Build	1993	Vista arriba y abajo con el ratón Etiquetas para tele-transportar personajes
Xngine	1995	3D real
Jedi	1995	Saltar y Agacharse Representación de los objetos mediante imágenes escaladas del objeto 3D real
Quake	1997	Optimización de renderizado Z-Buffering Aceleración por hardware Puntos de luz
RenderWare	1998	Modificación del arte y los procesos del juego en tiempo real.
idTech2/Quake2	1997	OpenGL Luminosidad de varios colores Librerías dinámicas en C Liberación de parte del código
GoldSrc	1998	OpenGL Direct3D
Unreal	1998	Edición de mapas Detección de colisiones Filtrado de texturas
Outcast	1999	*Voxel* Render a grandes distancias Bump-mapping Anti-Aliasing Sombreado dinámico Sistema de partículas Animación mediante esqueletos.
Quake3/idtech3	1999	Animación a partir de vértices Superficies curvas Colores de 32 bits Capacidades de red
GeoMod	2001	Modificaciones del terreno ante diferentes eventos
GameBryo	2003	Desarrollo multihilo Physix DirectX 9 y 10

		Compatibilidad con editores 3D
Source	2004	Iluminación Sombreados dinámicos Físicas Superficies acuáticas Optimizaciones multiprocesador Arquitectura de red eficiente
Euphoria	2008	Creación de animaciones a partir de simulación de esqueleto, músculos y colisiones.

Tabla 2. Evolución de los motores de juego

3.3.Estado actual de los motores de juego

La cantidad de motores de juego que existen hoy en día es enorme, existe una página web [19] que contiene una base de datos de motores de juego, y hasta el momento se han registrado 290 motores de juego distintos.

Medir la calidad de un motor de juego y crear un ranking de los mismos, es una tarea complicada, debido al gran número de motores que existen. No hay un criterio común para definir la calidad de un motor, pero algunas podrían ser, nivel de tecnología alcanzada, número de títulos publicados, facilidad de uso o compatibilidad con distintas plataformas. Pese a no haber un consenso en el método de evaluación, sí que existen varias revistas y páginas web que han hecho sus propios rankings del año 2009 y en las que coinciden algunos de los motores mencionados en las mismas.

Ranking de motores en el 2009		
Ranking develop-online [20]	**Ranking Ign [21]**	**Ranking NeoTeo [22]**
Torque 3D	RAGE Engine	Source Engine
Vicious Engine 2	CryENGINE 3	IW Engine
Bigworld Technology Suite	Naughty Dog Game Engine	id Tech 4
Vision Engine 7.5	The Dead Engine	RAGE
Infernal Engine	Unreal Engine	MT *frame*work
BlitzTech	Avalanche Engine	EGO Engine
Unity 3D	IW Engine	Geo-Mod Engine 2.0
CryENGINE 3	Anvil Engine	Anvil Engine
Gamebryo Lightspeed	EGO Engine	CryENGINE 3
Unreal 3	Geo-Mod Engine 2.0	Unreal 3

Tabla 3. Ranking de los motores de juego del 2010

Las tres listas de la tabla tienen discrepancias entre los motores de juego que consideran más potentes, pero algunos como el "*CryENGINE 3*" [23], "*Unreal 3*" [24] coinciden en las 3 listas, y otros como "*Anvil*", "*Geo-Mod Engine 2.0* ", "*EGO Engine*", "*IW Engine*" y "*RAGE*" [25], coinciden en dos de ellas. De la lectura de los artículos que definen un poco las características de los motores evaluados, se desprende que el criterio principal a la hora de juzgar los motores de juego, son los juegos que se crean con ellos. En los artículos se remarcan principalmente, tanto su éxito en el mercado, como su

calidad gráfica y animación, de modo que al final, el mercado parece ser el que pone a cada motor en su sitio, algunos de estos juegos que han tenido un gran éxito son "*Assasins Creed*" de "*Anvil Engine*", "*Grand Theft Auto*" de "*RAGE*", "*Far Cry 2*" y "*Crysis*" de "*CryENGINE 3*" o "*Gears of War*" y "*Mass effect 2*" de "*Unreal Engine 3*". Cada motor de creación de juegos es diferente en características a los otros. Además, dichos motores son creados con un objetivo específico. Si un programa diseñado para hacer juegos, no se usa para su propósito, entonces o no es un buen programa, o no se ha sabido demostrar su potencial, bien sea por dificultad, o por falta de recursos.

3.4.*Blender*

3.4.1. Descripción

Blender es una suite de herramientas integradas, que proporciona todo lo necesario para crear videos animados, y videojuegos mediante una interfaz gráfica que abstrae al diseñador de la programación, permitiéndole hacer todo de un modo mucho más visual. *Blender* es totalmente gratuito y de código libre, y sus características principales son:

Interfaz: *Blender* dispone de un interfaz creado a base de ventanas que no se solapan, y es totalmente configurable. Tiene capacidad para deshacer pasos en todos los niveles del interfaz, el espacio de ventanas es divisible en más espacios de ventana, y cada uno de ellos es configurable por separado. Además dispone de un editor de textos para tomar notas, escribir textos 3D o crear scripts en *Python*. Todo este Interfaz es consistente para todas las plataformas que soporta.

Modelado: *Blender* dispone de una herramienta de modelado muy potente, con capacidades similares a las de las mejores herramientas de pago. Dispone de todas las características necesarias para realizar un modelado profesional. Dispone desde las formas básicas típicas de cualquier editor de este tipo, hasta las más avanzadas técnicas de escultura, pasando por un gran número de modificadores, comandos y modos de edición, que son capaces de obtener resultados como el de la ilustración 18:

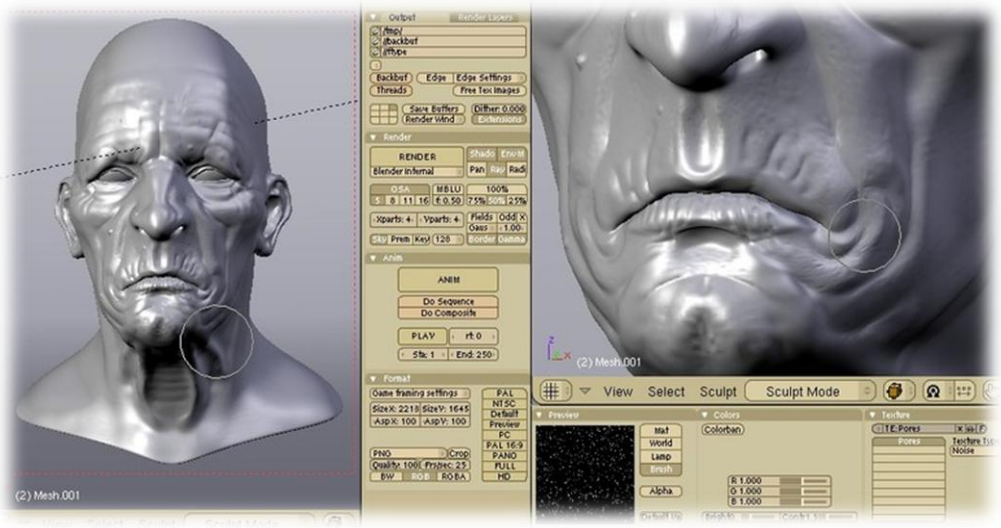


Ilustración 18. Ejemplo modelado *Blender*

***Rigging*:** El sistema de *rigging* de *Blender*, es un sistema avanzado utilizado por muchos de los editores 3D que existen hoy en día en el mercado. Permite crear esqueletos de un modo rápido y sencillo, y asignárselos a las mallas que van a controlar mediante distintas técnicas automáticas. También dispone de varias herramientas para corregir deformaciones no deseadas de la malla, provocadas por el movimiento del esqueleto, tales como la deformación volumétrica, o el uso de Cuaterniones.

Animación: La animación a partir de esqueletos y *"frames"* es una de las más utilizadas hoy en día. *Blender* proporciona un excelente interfaz para llevar a cabo este tipo de animación. Además dispone de herramientas como el "*IK chaining*", animación no lineal, (muy útil para poder mezclar animaciones individuales), creación de objetos nuevos a partir de otros existentes, o soporte para reproducción, mezclado y edición de audio. Una característica muy útil, es que se puede almacenar el *"frame"* que está recorriendo una animación, y acceder a él desde un script en *Python* para realizar acciones durante distintos momentos de la ejecución de dicha animación.

Mapeado de texturas UV: La herramienta de mapeado UV permite desenvolver una malla sobre una imagen, mediante distintos métodos de desenrollado. Posteriormente es posible mover los vértices de la malla desenrollada para ajustarlos al dibujo de la imagen de la textura. Dispone además de distintas opciones como la replicación o subdivisión de la textura además de otras opciones sobre texturas dinámicas.

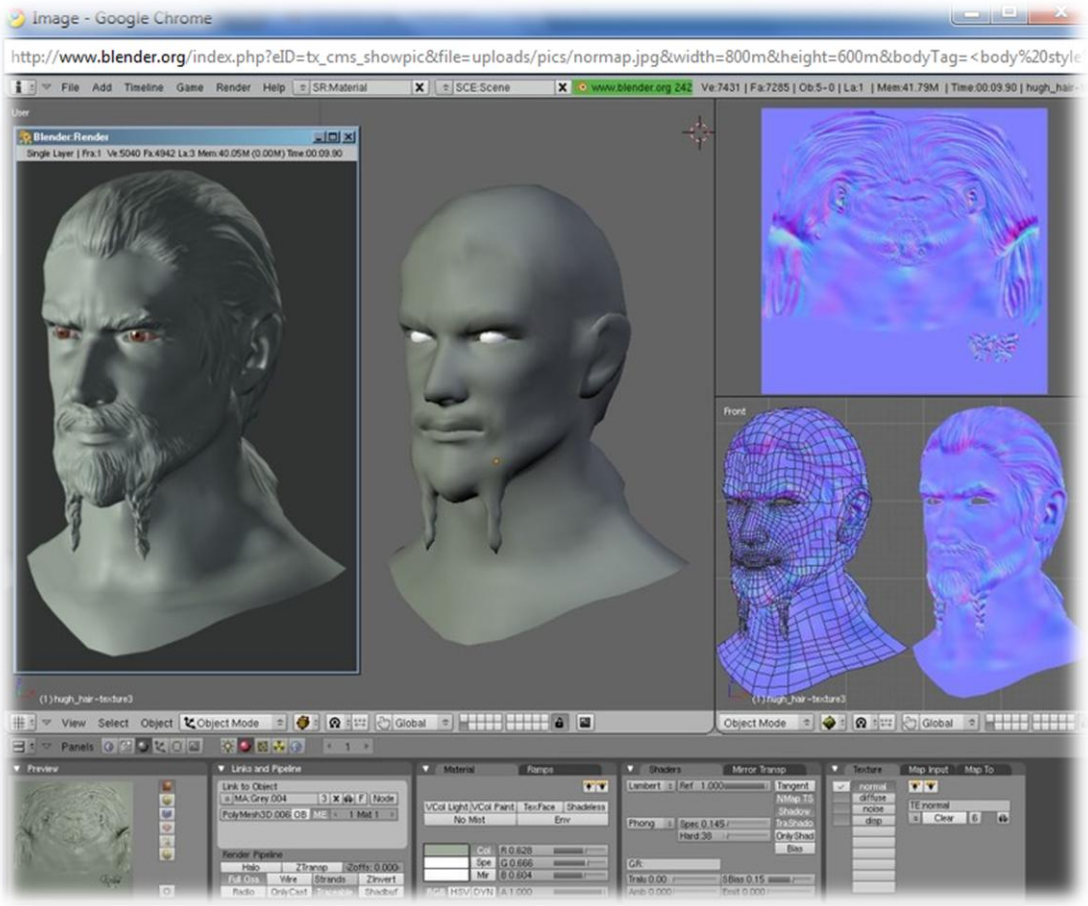


Ilustración 19. Ejemplo de mapeado de una textura UV para poner relieve a una cara.

Renderizado: *Blender* incluye un trazador de rayos integrado, aunque soporta muchos otros como PovRay [26], V-Ray [27] o Lux [28]. Realiza un renderizado por secciones, incluye capas y cache de renderizado y da soporte a efectos tales como halos, humo, destellos de luz, oclusión ambiental, desenfoque, *toon shading* [29] o radiosidad.

Shading [30]**:** Utiliza sombreadores difusos (*Lambert, Minnaert, Toon, Oren-Nayar*), y sombreadores especulares como *(WardIso, Toon, Blinn, Phong, CookTorr*). Dispone de un editor de nodos para la creación y mezcla de materiales, y en conjunto, un editor de materiales muy completo y sencillo de utilizar.

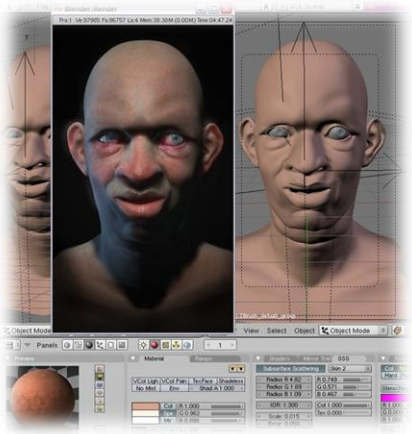


Ilustración 20. Ejemplo de edición de materiales

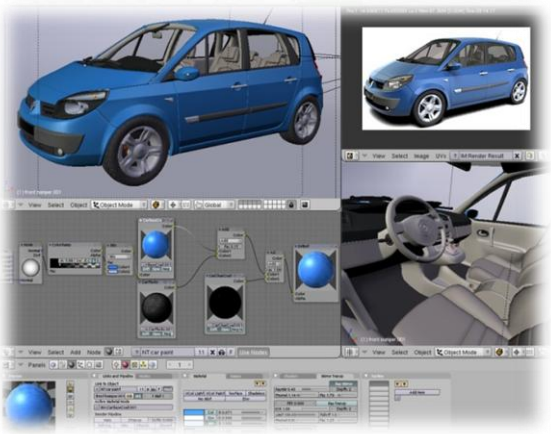


Ilustración 21. Editor de nodos de materiales

Físicas y partículas: *Blender* permite asignar sistemas de partículas a cualquier malla, o porción de esta, para simular cabello, espinas o humo a un alto coste computacional. Las partículas se pueden controlar mediante curvas, viento, vórtices y pintura de pesos.

Blender también permite simular fluidos, Cuerpos suaves y Cuerpos rígidos, y modificar sus propiedades físicas para obtener distintos efectos visuales. El motor de físicas incluye detección de colisiones, y efectos de campos de partículas, como viento y vórtices.

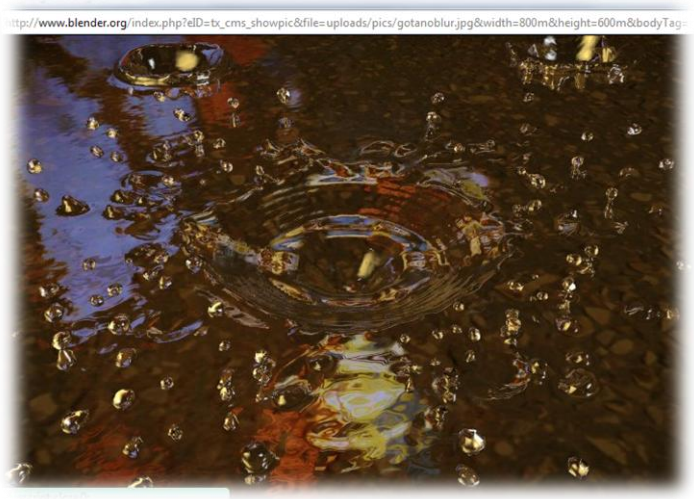


Ilustración 22. simulación de fluidos

Ilustración 23. partículas simulando plumón

3D en tiempo real para la creación de videojuegos: esta característica de *Blender* es la que más se va a desarrollar a lo largo del proyecto, y proporciona un interfaz gráfico para la inserción de lógica en los videojuegos sin necesidad de programar. Además ahora soporta la librería de físicas de libre distribución "*Bullet Phisics*" [31], que es la librería que utiliza la videoconsola de última generación PS3. Para un control más avanzado de los videojuegos, el interfaz gráfico permite

incorporar scripts en *Python* como controladores, y dispone de una librería "*GameLogic.py*" muy completa para la implementación de estos controladores.

Ilustración 24. Ejemplo de videojuego

Gestión de archivos: Con *Blender* se puede guardar toda la información de una escena en un archivo con extensión "*.blend*", incluyendo las imágenes utilizadas como texturas. Los ficheros de *Blender* proporcionan compatibilidad hacia adelante y hacia atrás, admiten compresión, encriptación, y pueden servir como librería para otros archivos "*.blend*". Además dan soporte a numerosos tipos de archivos 2D y 3D.

Soporte multiplataforma: *Blender* soporta las siguientes plataformas y mantiene su interfaz idéntico para cada una de ellas:

- Windows 2000, XP, Vista
- Mac OS X (PPC and Intel)
- Linux (i386)
- Linux (PPC)
- FreeBSD 5.4 (i386)
- SGI Irix 6.5
- Sun Solaris 2.8 (sparc)

3.4.2. *The Blender Foundation*, desarrollo y soporte

"*The Blender Foundation*", es una organización independiente, que actúa como una corporación pública sin ánimo de lucro que persigue los siguientes objetivos:

- Establecer servicios para usuarios activos de *Blender* y desarrolladores de *Blender*.
- Mantener y mejorar el producto *Blender* actual, mediante un sistema de código fuente público, con licencia GNU GPL.

- Establecer mecanismos de financiación que sirvan a los objetivos de la fundación y cubra sus gastos.
- Para proporcionar a la comunidad de internet, acceso a la tecnología 3D en general, con *Blender* como núcleo.

"*The Blender Foundation*" tiene sus oficinas centrales en Amsterdam, donde Ton Roosendaal, el jefe de proyecto, trabaja a tiempo completo junto con una pequeña plantilla, gracias a los ingresos recibidos de la venta en la tienda online, y de diversas publicaciones. Gracias a esos ingresos pueden también organizar actividades por ejemplo para *Siggraph* [32], apoyar proyectos de documentación y desarrollo, y mantener la contabilidad de la fundación.

Las principales actividades de *Blender* se agrupan en pizarras o proyectos que son los siguientes:

- **Desarrollo de software:** Disponen de un espacio [33], donde se hospeda el proyecto principal de *Blender*. Esto no impide que desarrolladores voluntarios creen ramas del proyecto con la función de investigar nuevas tecnologías y desarrollos para *Blender*.
- **Seguimiento de errores:** existe una guía de pasos a seguir a la hora de reportar un error, que son: asegurarse de que realmente es un error, evitar duplicados, simplificar, proporcionar una evidencia del error, y ser completo en la explicación del error. Todos estos pasos vienen explicados en la web de *Blender* [34].
- **Educación y entrenamiento:** Actualmente, se está tratando de crear proyectos de educación y entrenamiento oficiales para *Blender*.
- **Parches y Scripts:** En *Blender* admiten contribuciones voluntarias, en forma de parches que añadan funcionalidades a *Blender*. Para participar sólo hay que crear un archivo "*.blend*" o un script y enviarlo.
- **Desarrollo de documentación:** se ha creado un tablón en el que se puede escribir libremente documentación para los usuarios finales de *Blender*.
- **Investigación de funcionalidad**: para proporcionar *feedback* sobre el desarrollo, y tener en cuenta propuestas de los usuarios, se ha creado un tablón de Funcionalidad. Al que puede acceder cualquier usuario, habiendo creado previamente una cuenta.

3.4.3. "Yo Frankie", el videojuego de código abierto desarrollado en *Blender*

"*Yo Frankie"* [35] es un videojuego *Open Source* creado en *Blender*. El juego pertenece al género de plataformas, y en él se maneja a una ardilla, "*Frankie*", con la que se van recorriendo los distintos mapas hasta llegar al final de cada nivel. "*Frankie*" puede saltar, planear, caminar, correr, agarrarse a salientes, ahogarse en el agua, y quemarse en la lava. También puede interactuar con los enemigos que se puede encontrar en cada pantalla, atacándolos, levantándolos, trasportándolos y tirándolos.

Además el juego permite partidas multijugador en una pantalla partida, manejando a "*Frankie*" y a un segundo personaje que es un mono, utilizando las teclas de un mismo teclado para controlarlos, y que pueden cooperar para superar los distintos niveles.

Ilustración 25. Ejemplo1 Yo Frankie

Ilustración 26. Ejemplo2 Yo Frankie

Este juego es una de las razones que me ayudó a seleccionar *Blender* como el motor de juego, ya que proporciona una idea del nivel profesional que se puede llegar a alcanzar con *Blender* a la hora de desarrollar videojuegos. y de que el motor tenía unas capacidades de desarrollo muy elevadas. Además, al ser un juego de código libre, se puede consultar su código para aprender a utilizar *Blender*.

3.5. Conclusiones

Como conclusiones del estado del arte expuesto, podemos deducir las siguientes:

1- Fijándonos en la historia de los motores de juego, y en el estado actual de los mismos, es un hecho que son herramientas útiles y utilizadas para crear videojuegos.
2- Se invierte en su desarrollo y actualización, y por tanto es útil conocer su funcionamiento.
3- *Blender* es un motor de juego con el que perfectamente se pueden desarrollar proyectos profesionales, que tiene un equipo de desarrollo detrás que lo mejora día a día.
4- Al ser de código abierto, admite modificaciones y mejoras aportadas por cualquier usuario interesado, lo cual posibilita una evolución más rápida del motor.

Existen varias razones por las que se eligió *Blender* para realizar este proyecto:

1- Es un software de código abierto y gratuito, con lo cual si el tiempo de desarrollo se alargaba demasiado, no sería necesario pagar una licencia para poder utilizarlo.
2- El diseño 3D y el motor de juego, son un mismo programa, y no se producen incompatibilidades a la hora de tener que importar modelos, ya que esta operación no es necesaria.
3- Existe muchísima información, ejemplos y tutoriales en internet que serían muy útiles a la hora de la creación del manual de *Blender*.
4- Complementando a la Tercera causa, se puede afirmar que no existe organización y falta completitud en los tutoriales y ejemplos mencionados. Los ejemplos, guías y tutoriales encontrados en Internet eran normalmente sobre un único concepto referente al motor y

no se encontraba ningún manual que explicase la secuencia de pasos a seguir para crear un juego avanzado con referencias detalladas al interfaz de *Blender*.

5- La calidad, tanto técnica como gráfica, que se puede alcanzar al desarrollar un videojuego con *Blender*. La prueba es el videojuego "*YoFrankie"*, expuesto anteriormente.

6- Dispone de un modo de diseñar juegos innovador y original. La lógica se incluye en los juegos mediante un sencillo interfaz gráfico, a partir de sensores, controladores y actuadores, sin necesidad de programar nada. Además de mediante el interfaz gráfico, también se pueden configurar los controladores utilizando scripts en *Python* para realizar acciones más complejas.

4. Manual de desarrollo de videojuegos en *Blender*

4.1.Introducción a *Blender* Game Engine

Blender está formado por dos componentes principales, integrados en un único interfaz. Por una parte es una herramienta para la creación de películas o cortos de animación en 3D, y por otra, es un motor de juego con el que se puede aplicar lógica a objetos creados en 3D. Puesto que el objetivo de este proyecto es crear un manual que sirva como guía para poder realizar un videojuego en *Blender*, muchas de las funcionalidades del programa que sólo aplican a la parte de animación y renderizado, quedarán fuera de dicho manual.

En este apartado se explicará en primer lugar un método general para la creación de videojuegos, definiendo varios pasos ordenados a seguir.

Después se hará una descripción general del interfaz, para familiarizar al lector con los términos y las pantallas más comunes de *Blender*, a los que se hará referencia a lo largo del resto de manual. Esta introducción ayudará a agilizar el desarrollo de las demás páginas del manual.

Por último se expondrán detalladamente las distintas técnicas existentes en *Blender* para desarrollar videojuegos, como la animación de personajes, creación de materiales, introducción de lógica, además de algunos conceptos sobre diseño 3D.

4.1.1. Pasos para crear un juego avanzado en *Blender*

En este apartado se enumeran los pasos básicos y el orden de los mismos, necesarios para realizar un videojuego en *Blender*, y se resumen muy brevemente las distintas opciones que se pueden seguir para llevar a cabo cada uno de ellos.

1. Creación u obtención de los personajes principales del videojuego:

Cualquier juego necesita de un personaje que el usuario pueda manejar, ya sea una persona, un vehículo, un ejército, o una bola como puede ser en el caso del videojuego "*Pinball*". Uno de los puntos fuertes de *Blender* es la herramienta de diseño 3D que proporciona, una herramienta que permite crear desde objetos básicos como esferas o cubos, hasta mallas que pueden representar perfectamente cualquier objeto real. Es cierto que el diseño de personajes en 3D supone un trabajo bastante costoso, y conseguir resultados realistas es muy difícil, así que para empezar a diseñar un juego y conocer las capacidades de Bleder, se puede empezar por algo sencillo como una esfera que recorra mapas, o si se quiere empezar con algo más complejo, se puede descargar de internet uno de los muchos modelos gratuitos, disponibles en diversas páginas web, tales como el repositorio oficial de *Blender* [36].

A continuación se muestran 2 ejemplos de personajes creados en *Blender*, el primero creado por el autor del este proyecto, y el segundo obtenido del repositorio oficial de *Blender*.

Ilustración 27.modelo creado por el autor del proyecto

Ilustración 28. modelo descargado del repositorio oficial de *Blender*

2. Creación de un esqueleto para el personaje:

Para poder mover al personaje de un modo sencillo, se debe crear un esqueleto hueso a hueso. Cuantos más huesos tenga el esqueleto, movimientos más complejos y realistas se podrán realizar. Para un personaje humano, se deberá crear un esqueleto similar al esqueleto de una persona real, pero muy simplificado, sólo con los huesos que lleven a cabo movimientos visibles. Por ejemplo, si queremos que se vea abrir y cerrar la mano, habrá que hacer un hueso para la mano y uno para cada falange y dedo, en cambio si no vamos a cerrar las manos del personaje, o no van a estar visibles, con un hueso para toda la mano incluidos los dedos, será suficiente. Si en cambio el personaje que hemos creado es un coche, será mucho más sencillo crear huesos para el mismo ya que sólo se necesitará un hueso para controlar el movimiento de la carrocería y otros cuatro para simular el giro de las ruedas.

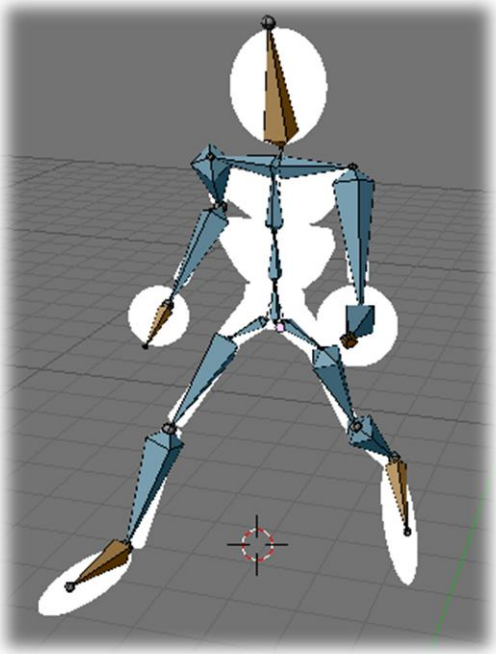


Ilustración 29. Ejemplo de esqueleto con 20 huesos

3. Creación de un escenario básico:

   Todo juego necesita un lugar, escena o escenario en el que desarrollarse, ya sea un ring de lucha, un campo de batalla entre dos ejércitos, o una pista de carreras. Lo mejor es empezar por diseñar un escenario lo más simple posible, con los objetos más representativos del mismo con los que deba interactuar el personaje, sin preocuparse de momento de texturas ni de detalles de gráficos. Por ejemplo un plano para un ring o la trayectoria de una pista de carreras y los muros contra los que podría chocar el coche serían 2 buenos escenarios iniciales. Una vez esté acabado el juego, añadir más escenarios y más detalles a estos, será una mera cuestión de diseño y modelado en 3D.

4. Generación de las animaciones propias del personaje:

   Personalmente la cantidad y calidad de las animaciones es uno de los factores que más influyen en mejorar la calidad y el realismo de un juego. Por lo tanto, esta fase de desarrollo es muy importante y no debe importarnos invertir una buena cantidad de tiempo en realizar esta tarea. Para llevarla a cabo en *Blender*, se utilizarán los esqueletos creados anteriormente, que nos permitirán mover la malla del personaje en cuestión, y almacenar las acciones creadas con un nombre para ser ejecutadas posteriormente según sean requeridas por la lógica del juego.

5. Implementación de la lógica propia del personaje y de los objetos con los que interactúa:

   Es en esta fase en la que el ingeniero debe poner toda su imaginación, y destreza al programar para decidir que va a ocurrir cuando se pulse una tecla, se detecte una colisión con cierto objeto

o con otro personaje, o en general como va a reaccionar el personaje cuando ocurra algún evento relevante en su entorno.

En esta fase se debe tener muy claro que es lo que se quiere que sea capaz de hacer el personaje ya que si no es así, se puede caer en un bucle sin fin en el que cada vez se añaden más funcionalidades al mismo, normalmente debidas al entusiasmo del autor del juego. Una vez se conozcan a la perfección las capacidades del personaje se deberá pensar el modo más eficiente y sencillo de programarlas.

En *Blender*, como veremos más adelante para implementar la lógica de cada objeto, se utilizan sensores que detectan las interacciones con el entorno y la interfaz de usuario, controladores que implementan la lógica del juego en sí, y los actuadores que ejecutan las acciones previamente creadas, modifican variables y en definitiva actúan sobre el entorno del personaje y sobre él mismo.

Hay que tener en cuenta que cuanto más independiente sea la lógica entre los distintos objetos, más fácil será integrarlos en el juego y hacer modificaciones al mismo.

6. Implementación de la lógica global del juego:

Una vez definidas las capacidades de los objetos presentes en el juego, será necesario definir las normas generales del mismo. Estas normas o reglas serán por lo general las condiciones de victoria o derrota, disparadores que cambien el escenario, el final de una pantalla, el tiempo de juego transcurrido, etc.

Un modo sencillo de implementar esta lógica global del juego en *Blender*, es crear un objeto invisible en el juego, capaz de recibir mensajes del resto de objetos mediante sensores de tipo mensaje, y en función de los mensajes que reciba, actúe en consecuencia. Por ejemplo, si en un juego de lucha el jugador número uno se queda sin vida, este enviará al objeto controlador global un mensaje indicando esta información, y el controlador activará un actuador que muestre de algún modo al jugador 2 como victorioso, ya sea con un letrero en 3D o haciendo que la cámara gire alrededor de él. Posteriormente deberá cargar el menú principal, el menú de selección de personajes o simplemente pasar a otro escenario con el siguiente luchador al que se debe derrotar.

7. Diseño e implementación de los detalles del juego:

En esta fase de diseño es donde el programador puede empezar a mejorar la calidad del juego hasta donde él quiera, creando texturas detalladas para los objetos, efectos especiales como bolas de fuego o efectos luminosos y sonoros que incrementen el realismo de la situación.

Puesto que en esta etapa ya debería estar acabado el juego en sí mismo, el diseñador puede recrearse añadiendo todos los detalles que quiera, refinando animaciones, mejorando los escenarios, teniendo sólo en cuenta el plazo de entrega que tenga marcado. Personalmente es una de las fases más gratificantes del desarrollo, ya que una idea a la que se ha ido dando forma poco a poco, ahora se la moldea para que sea cada vez más real y entretenida.

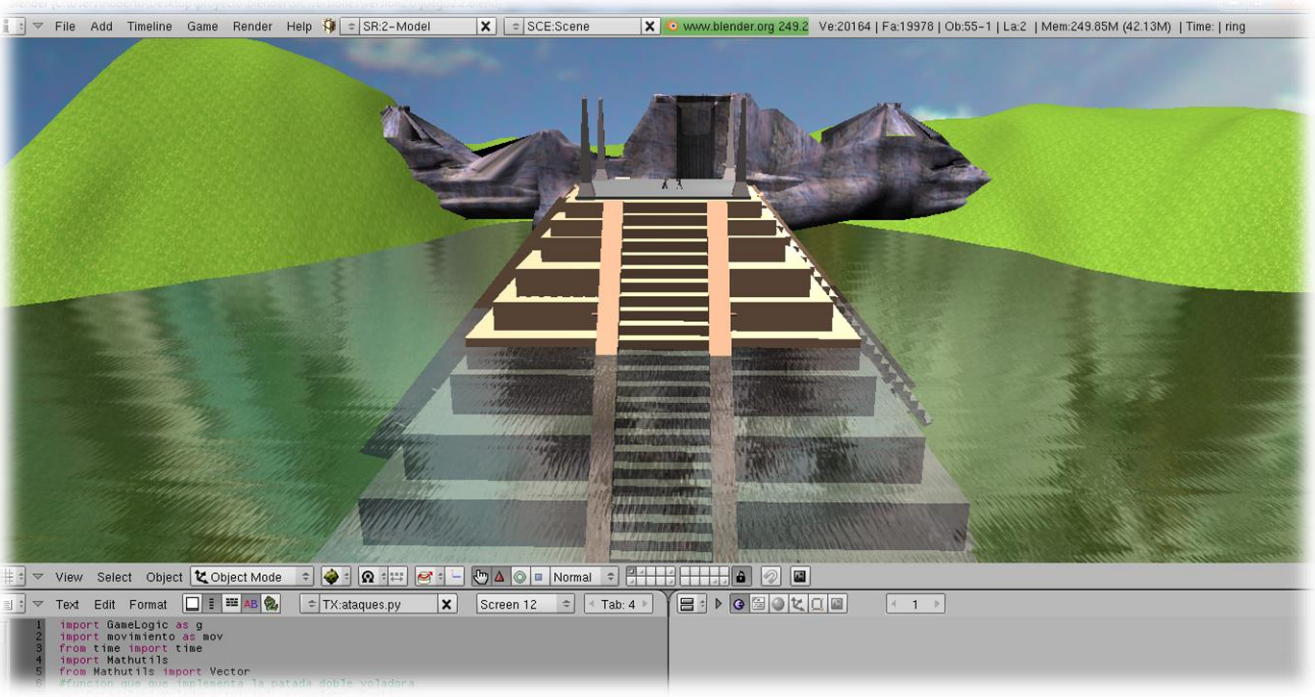


Ilustración 30. Ejemplo de entorno detallado

En la imagen superior se puede ver un ejemplo de un entorno 3D detallado que dispone de un templo maya sumergido en un lago, con un ring de combate en su parte superior, detrás del cual se aprecian unas montañas divididas por una catarata. Además el escenario está cubierto por una cúpula celeste.

4.1.2. Resumen Descriptivo del interfaz

En este subapartado se definen las principales pantallas y conceptos que se deben conocer de *Blender* para poder implementar juegos, y entender las instrucciones que se irán dando a lo largo del resto del manual, instrucciones como "*se abre la ventana x*" o "*se crea una nueva escena*".

Este apartado se dividirá para su mejor comprensión en una sección que describa el concepto de escenas y capas, y otra sección mucho más extensa que describa las pantallas y sus componentes más necesarios para la creación de juegos.

Hay que destacar como característica más representativa de la interfaz de *Blender*, que se trata de una interfaz divisible y replicable, es decir que cada pantalla del interfaz se puede subdividir en 2, y lo mismo pasa con cada una de estas subdivisiones, pudiendo así obtener las pantallas que el programador necesite para diseñar de una manera más cómoda su videojuego.

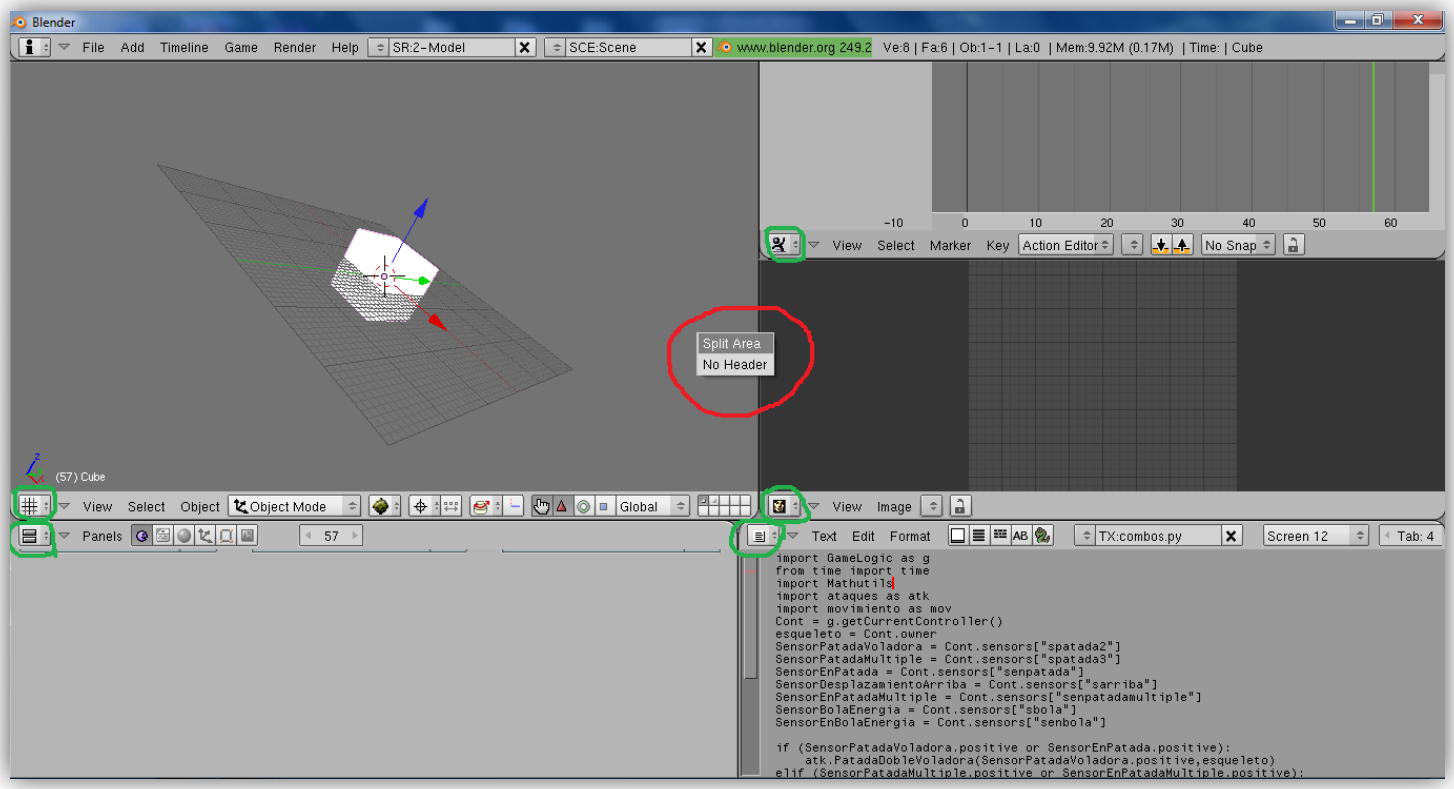


Ilustración 31. Interfaz divisible y replicable

En la imagen superior se ve un interfaz dividido en 5 pantallas. Los menús desplegables rodeados con círculos verdes, muestran el tipo de pantalla que se tiene abierta y permiten cambiar este tipo con sólo pulsar sobre el menú desplegable. Con el círculo rojo se ha marcado el menú emergente que aparece al hacer clic derecho sobre una de las divisiones que existen entre dos pantallas, y sobre el que aparece la opción "*Split Area*" que nos permitirá dividir una pantalla de las existentes en otras dos.

Escenas (SCENES) y capas (LAYERS)

1. Escenas:

Las escenas en *Blender* pueden tener varias utilidades, pero la ventaja principal que se puede obtener de las escenas, es que proporcionan independencia para las distintas partes del juego.

Un ejemplo práctico del uso que se les puede dar a las escenas, es la creación de menús. Una escena puede ser el menú principal del juego, y cuando se selecciona una opción, se abre una nueva escena que puede ser el menú de configuración, de selección de personajes, de selección de partidas guardadas, etc. Una escena puede ser incluso las pantallas en las que vaya a jugar el usuario, teniendo almacenada una pantalla por escena, haciendo así muy sencilla la carga de los distintos niveles.

En la ilustración 32 se muestra como crear una nueva escena:

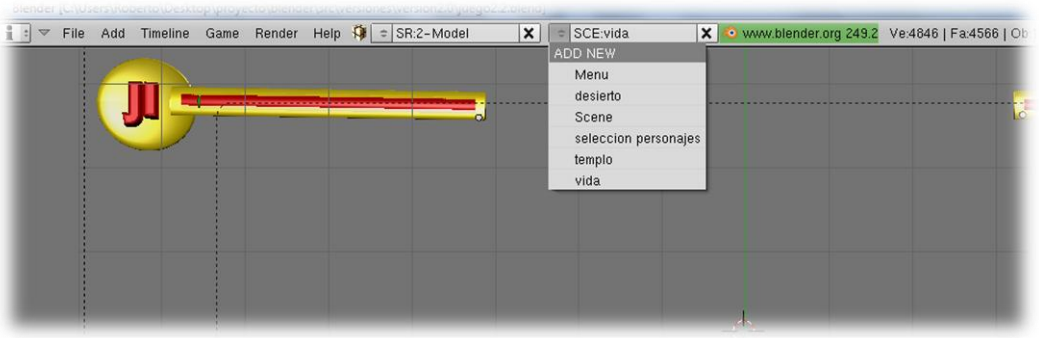


Ilustración 32. Creación de nuevas escenas

Como se puede observar en la ilustración 32, en la ventana principal de *Blender* existe un menú desplegable que contiene las letras "*SCE:*" seguidas del nombre de la escena actual. Para crear una nueva escena hay que abrir dicho menú desplegable y hacer clic en el primer elemento del menú que tiene escrito el texto "*ADD NEW*"

El hecho de utilizar escenas evita tener que estar cargando y eliminando de la pantalla los objetos y modelos uno a uno facilitando así la labor de programación y simplificando la lógica del programa. Al poder cargar una escena completa que incluye sus objetos y su lógica, se evitan posibles errores ya que una escena siempre se carga y se inicializa del mismo modo, el que haya definido el programador.

2. Capas:

Las escenas en *Blender* están compuestas de varias capas de visibilidad, de modo que si cualquiera de dichas capas está activa, los objetos que contenga serán los que se vean en pantalla al iniciar el juego. Al igual que las escenas, las capas, sirven para conseguir una mayor organización en el desarrollo del juego, pudiendo tener separados en diferentes capas los objetos de una escena basándose en cualquier tipo de criterio. Se pueden decidir tener en una capa los personajes, en otra el escenario básico, en otra los personajes hostiles, etc.

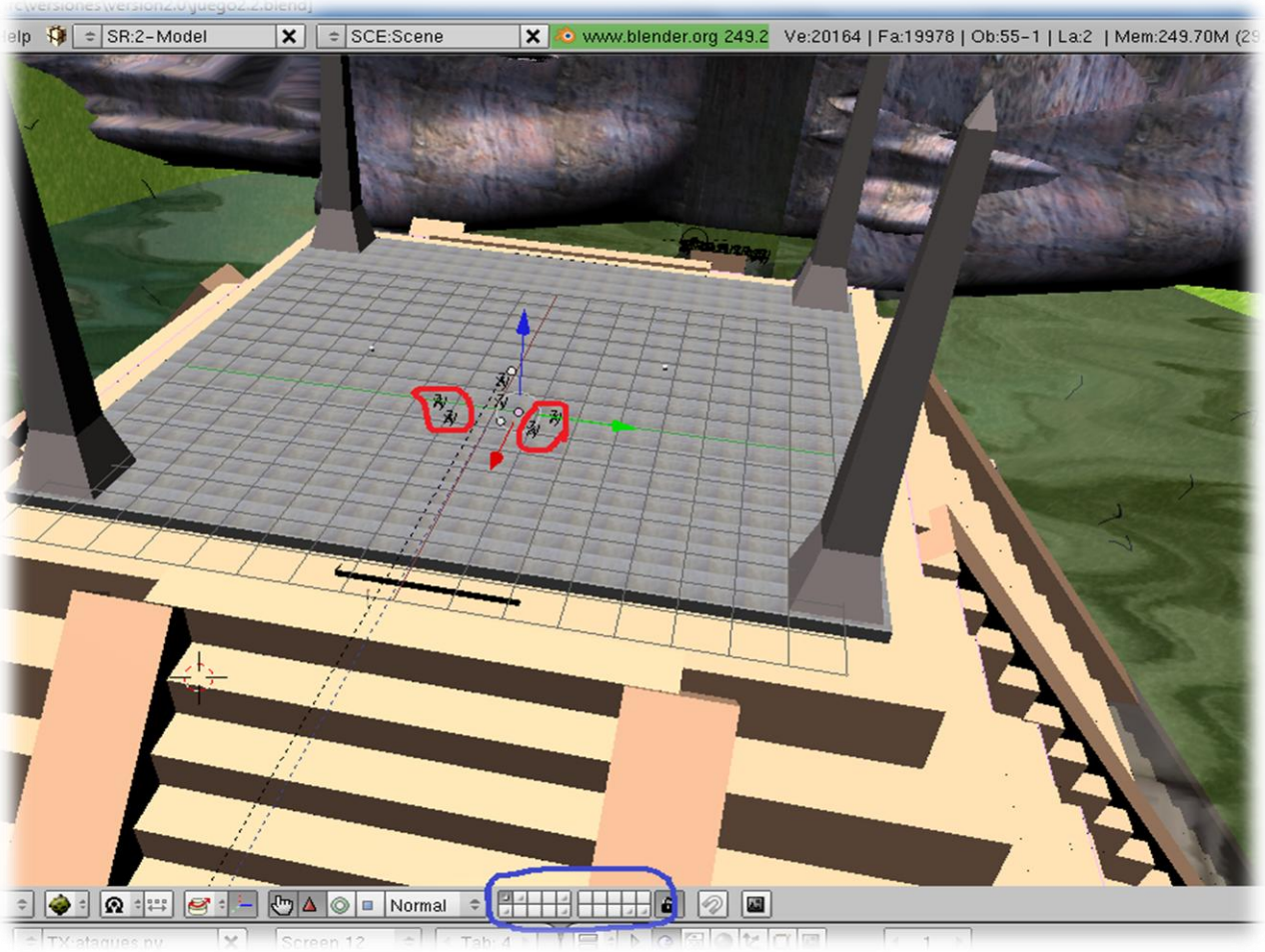


Ilustración 33. Capas en *Blender*

En la imagen superior se ha rodeado con un círculo de color azul, en la parte inferior del interfaz, una serie de botones que representan las 20 capas disponibles en cada escena. La primera capa esta sombreada en gris ya que es la capa activa, la que contiene el escenario que se ve en la imagen. Todas las capas que tienen un punto en el botón que las representa, contienen algún objeto, y el resto están vacías.

Pero las capas en *Blender* tienen una función mucho más importante que la mera organización de objetos. Son las capas las que permiten incluir en el juego objetos de un modo dinámico. Para entender esto mejor, es necesario saber que los objetos que queremos usar en un archivo creado en

Blender, deben existir en dicho archivo en una capa oculta. Es decir, que si tenemos un escenario en una capa visible y el jugador puede elegir por ejemplo entre 10 personajes distintos para poder jugar, todos esos personajes deben existir en las capas ocultas de la escena en cuestión. Siguiendo con el ejemplo, para hacer que el personaje seleccionado por el jugador aparezca en el escenario, será necesario un objeto de tipo "*empty*", rodeado con círculos rojos en la imagen anterior, con un actuador de tipo "*Edit object*" que será el que añada al objeto con el nombre indicado en los parámetros del actuador, en este caso, el personaje seleccionado.

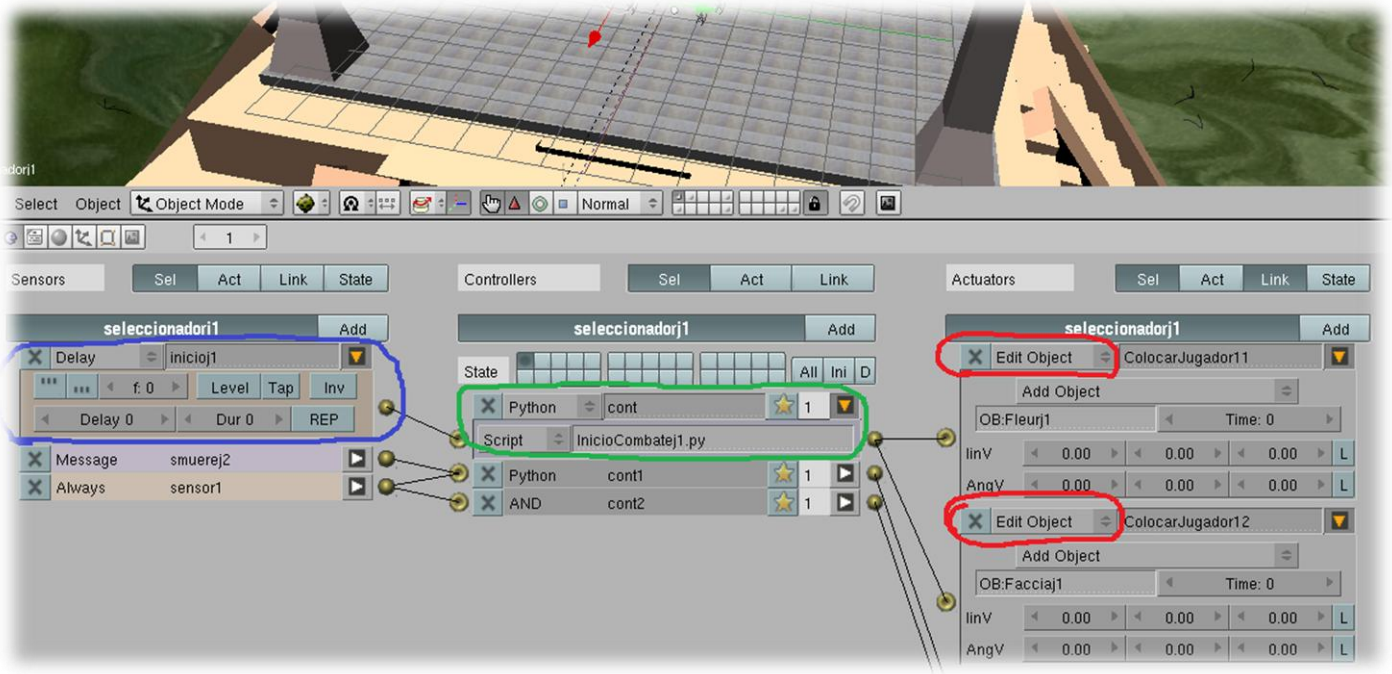


Ilustración 34. Inclusión de objetos dinámicamente

Lo que se muestra en la imagen superior, son los bloques lógicos de un objeto de tipo "*empty*", que se encuentra en la capa visible del juego, y que es el encargado de añadir al escenario los luchadores seleccionados por el jugador cuando se inicia el juego. El sensor rodeado en azul detecta un retraso de 0 segundos, es decir que nada más empezar el juego, activará el controlador rodeado en la imagen en verde.

El controlador es un script en Python, que leerá de un fichero el personaje seleccionado por el jugador 1 en el menú de selección de personajes, y dependiendo de si el personaje seleccionado es "*Fleur*" o es "*Faccia*", activará uno de los 2 actuadores que en la imagen se pueden ver rodeados en rojo. Los actuadores son de tipo "*Edit Object*" como se dijo con anterioridad, en su modalidad "*Add Object*", añadir objeto. Estos actuadores lo que hacen es cargar en la capa visible un objeto determinado por el parámetro "*OB:*", que se encuentre en una de las capas ocultas. Este concepto se explicará con más detalle en el punto "*4.7 Inserción dinámica de objetos en escenas*".

Descripción de las ventanas de Blender

Blender divide su interfaz en ventanas, a las que se puede acceder desde un menú desplegable. Cada una de estas ventanas contiene una funcionalidad de *Blender*, y todas las herramientas necesarias para sacarle el mayor partido a dichas funcionalidades. En este apartado se explica de un modo muy básico cada una de estas ventanas, para familiarizar al lector con el interfaz de *Blender*, y poder hacer referencia a estos conceptos posteriormente en el desarrollo detallado del manual.

1. 3D Window:

Esta ventana es en la que se diseñan y representan todos los objetos 3D del juego. En ella se pueden añadir formas básicas como cubos o esferas, y otras más complejas como esqueletos y curvas. Todos los objetos representados en esta ventana se pueden editar, escalar, rotar, desplazar, deformar y aplicar transformaciones booleanas. En general se les puede transformar casi de cualquier forma.

En los menús que se pueden ver en la parte inferior de la ilustración 35, se puede cambiar la vista que tenemos del entorno entre una vista ortográfica y una vista en perspectiva, y entre vista global y local (de un único objeto). Con el resto de botones y desplegables de la barra inferior se pueden aplicar las distintas transformaciones a los objetos 3D, se puede cambiar la visibilidad de los objetos, el modo de edición en el que estamos trabajando, y las capas que hay visibles en cada momento.

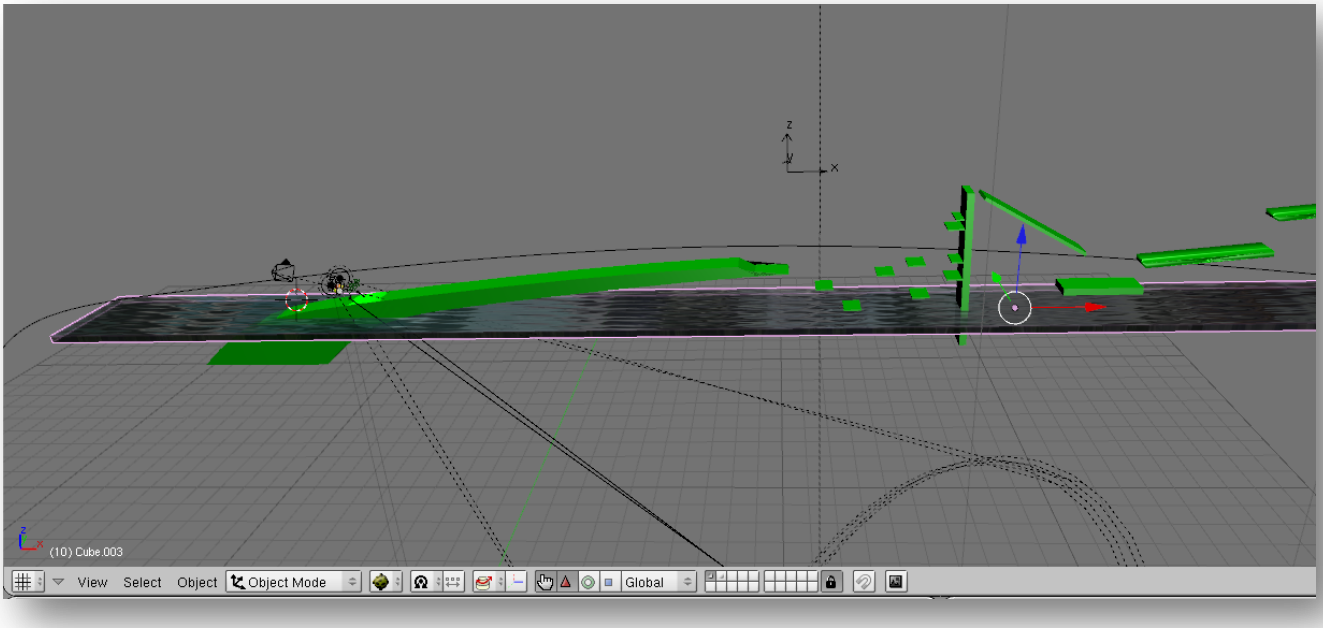


Ilustración 35. *3D Window*

2. IPO Curve Editor:

En esta ventana se pueden generar animaciones para un objeto, a partir de transformaciones sencillas. Como se puede ver en la ilustración 36, la pantalla consta de una línea temporal, en la que podemos almacenar en *"frames"* determinados las transformaciones aplicadas a un objeto en el momento que estas son almacenadas. De este modo si almacenamos 2 transformaciones distintas separadas por varios *"frames",* se creará una animación sobre el objeto, que lo irá transformando desde su forma inicial, hasta su forma final.

Pulsando la tecla "i", aparecerá el menú desplegable que permite elegir entre las tres transformaciones básicas existentes, y al seleccionar una de ellas, se guardará dicha transformación en el *"frame"* en el que se esté situado en la línea temporal. Estas transformaciones son desplazamiento, rotación y traslación.

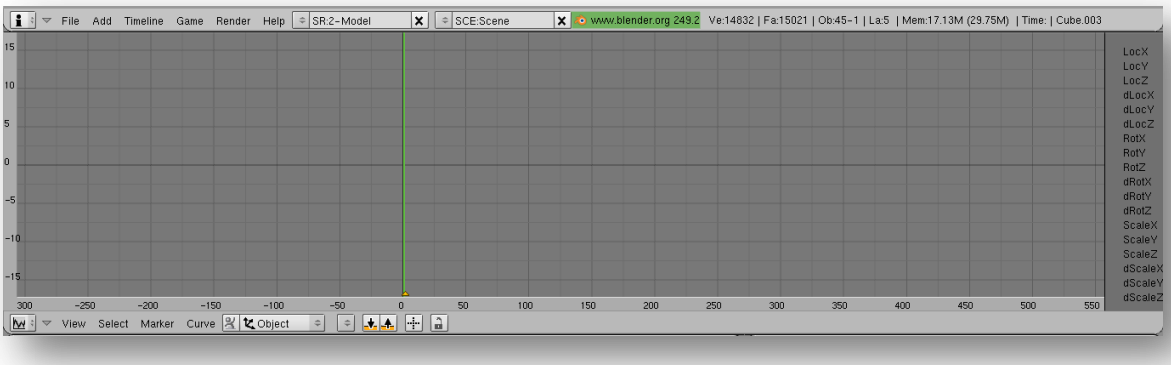


Ilustración 36. *IPO Curve Editor*

3. Action Editor:

En esta ventana se pueden crear animaciones de un modo similar a como se hace en la ventana "*IPO Curve Editor", pero* para objetos y transformaciones más complejas. Por ejemplo se puede crear una animación para un personaje humano, que represente la acción de andar. Si nos fijamos, esta animación requiere deformar la malla del personaje para poder mover las piernas del mismo. Para realizar esto, se necesita un esqueleto unido a la malla 3D.

Una vez se tiene la malla asignada a un esqueleto, se pueden rotar, trasladar y escalar cada uno de sus huesos, y almacenar dichas transformaciones en uno o varios *"frames"*. Una vez almacenadas 2 posiciones en 2 *"frames"* distintos, las transformaciones intermedias se generarán automáticamente. De nuevo, el modo de almacenar las transformaciones de cada hueso seleccionado en los distintos *"frames"*, es pulsando la tecla "i", y en el menú desplegable que aparecerá, seleccionar la transformación que se quiere almacenar. Las transformaciones almacenadas en la línea temporal se podrán posteriormente escalar, trasladar, copiar y aplicar numerosas transformaciones sobre las mismas.

Ilustración 37. *Action Editor*

Viendo la imagen superior se pueden ver dos iconos en color amarillo con los que se pueden copiar y pegar las claves seleccionadas de la línea temporal. A su lado está el menú desplegable en el que se dará nombre a la acción creada, necesario ya que un mismo esqueleto puede almacenar varias animaciones.

4. UV Image Editor:

En esta ventana se pueden importar imágenes, y utilizarlas como texturas sobre la superficie de una malla. Las imágenes se importan desde el menú "*Image*" que aparece en la parte superior de la ilustración 38, y las imágenes abiertas de este modo, se pueden empaquetar dentro del archivo de *Blender* para facilitar su transporte. Esta ventana también permite editar las imágenes importadas, pintando sobre ellas, recortándolas o escalándolas. Además al asignar una de estas imágenes como textura, se le pueden aplicar varios efectos, como aplicar la textura en modo mosaico, o dividiéndola y replicándola, y haciendo que la textura se vaya moviendo a lo largo del tiempo.

Ilustración 38. *UV Image Editor*

5. Audio Window:

Esta ventana permite editar ondas de audio, pero en este manual sólo se utilizará para cargar sonidos con extensión .wav en el archivo *Blender*. Estos archivos posteriormente podrán ser utilizados en los bloques lógicos para reproducir sonidos, al detectar una colisión, dar un salto o simplemente para reproducir una música de fondo en el juego.

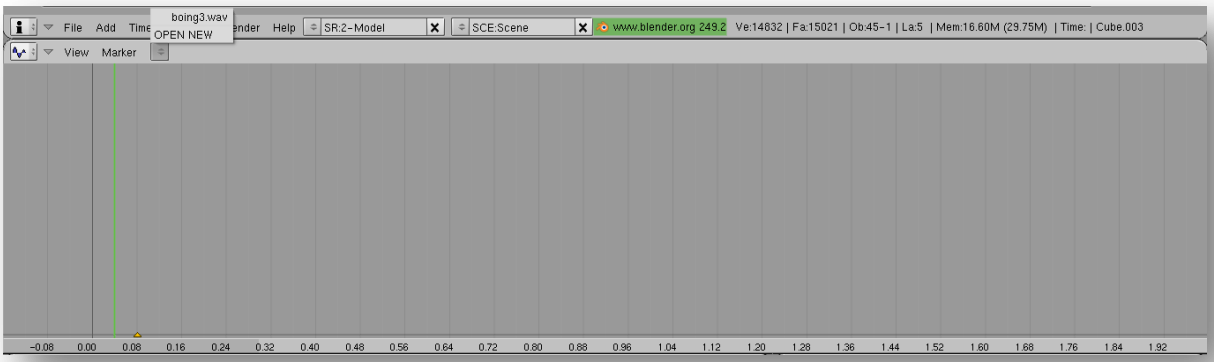


Ilustración 39. Audio Window

6. Text Editor:

En esta ventana es en la que se pueden crear nuevos scripts, utilizables para modificar la lógica del videojuego. El editor dispone de las herramientas básicas de un editor de textos de programación, tales como marcado de lenguaje, numeración de líneas y búsqueda y remplazo de palabras además de otras. Pero este editor no sólo sirve para crear scripts. Los textos escritos en este editor se pueden convertir en textos 3D que aparecerán como un objeto 3D en la ventana "*3D Window*", mediante el menú "*Edit*" que aparece en la parte superior de la ilustración 40.

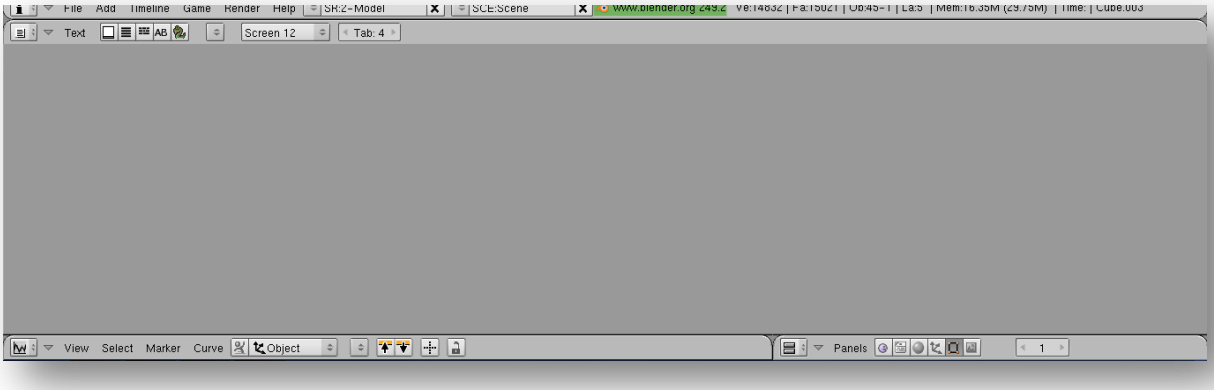


Ilustración 40. Text Editor

7. Buttons Window:

Esta es la ventana más completa y compleja de todas las que hay, y es que esta venta realiza muchas acciones distintas. Está dividida en varios sub-apartados a los que se puede acceder a través de los iconos que aparecen en la parte superior de la ilustración 41.

El primero de estos iconos lleva al sub-apartado en el que se introduce la lógica a los objetos 3D, y en el que se definen las propiedades físicas de estos objetos en caso de que no sean estáticos. Este apartado también permite definir si un objeto estará sometido o no a las fuerzas de la naturaleza simuladas por *Blender*.

El tercer icono es una bola roja, y es el encargado de crear los distintos materiales que se usarán en el videojuego, así como las texturas que definen dichos materiales. En este sub-apartado se podrá modificar color, rugosidad, luminosidad, reflexión y refracción de la luz sobre ese material o la luz que desprende el objeto en caso de tratarse de un punto de luz. Las texturas que forman parte de ese material también se crean en este sub-apartado, y Además, para cada una de ellas, permite elegir como se va a aplicar sobre la superficie del objeto sobre el que se mapea.

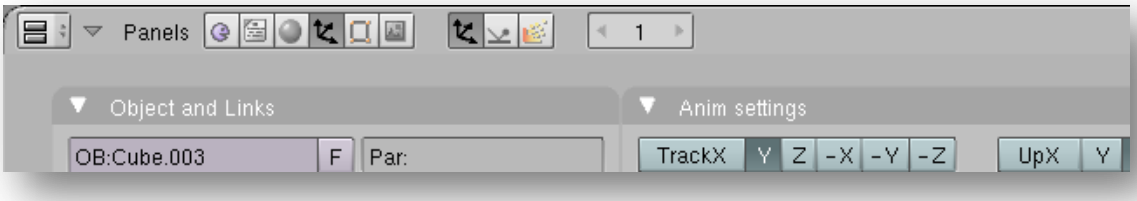


Ilustración 41. Buttons Window

El cuarto icono representa las propiedades del objeto seleccionado, y como se puede ver en la en la ilustración 41, está dividido a su vez en otros tres sub-apartados. El primero de estos sub-apartados permite realizar acciones sobre el objeto, que tienen que ver con la animación y el dibujado por pantalla del mismo, y además permite añadirle un buen número de restricciones, como límites de localización o caminos a seguir. El segundo de estos sub-apartados permite modificar parámetros referentes a las propiedades físicas del objeto en caso de tenerlas, y el último permite crear sistemas de partículas, más útiles para crear un video que un juego, debido a su alto coste computacional.

El quinto icono permite realizar operaciones sobre la malla de un objeto, crear grupos de vértices, cambiar el nombre del objeto y de la malla, y aplicar modificadores, que como veremos más adelante en el manual, serán muy útiles a la hora de diseñar los personajes y escenarios. Algunos de estos modificadores son la subdivisión suave de superficies, deformación de mallas, creación de una malla espejada, o aplicación de operaciones booleanas entre dos objetos.

4.2.Modelado de personajes y entornos

Este apartado se va a centrar en el diseño en tres dimensiones de objetos y personajes en *Blender*, pero puesto que un manual de diseño 3D podría tener una extensión superior a la del propio proyecto actual, sólo se darán algunas indicaciones sobre los comandos proporcionados por *Blender* más útiles e importantes respecto al diseño y creación de objetos. Se explicará cómo crear un esqueleto, que es algo imprescindible para la posterior creación de animaciones, y como vincularlo a la malla que se desee animar. También se explicará brevemente como asignar texturas de tipo UV a las mallas, junto con alguna técnica especial para crear materiales realistas.

4.2.1. Algunas directrices sobre el diseño 3D en *Blender*

Este apartado no pretende formar a un experto en diseño 3D. Lo que se pretende con esta parte del manual, es que el programador adquiera los conocimientos básicos sobre los comandos y el uso del interfaz, que serán imprescindibles a lo largo de todo el desarrollo del juego.

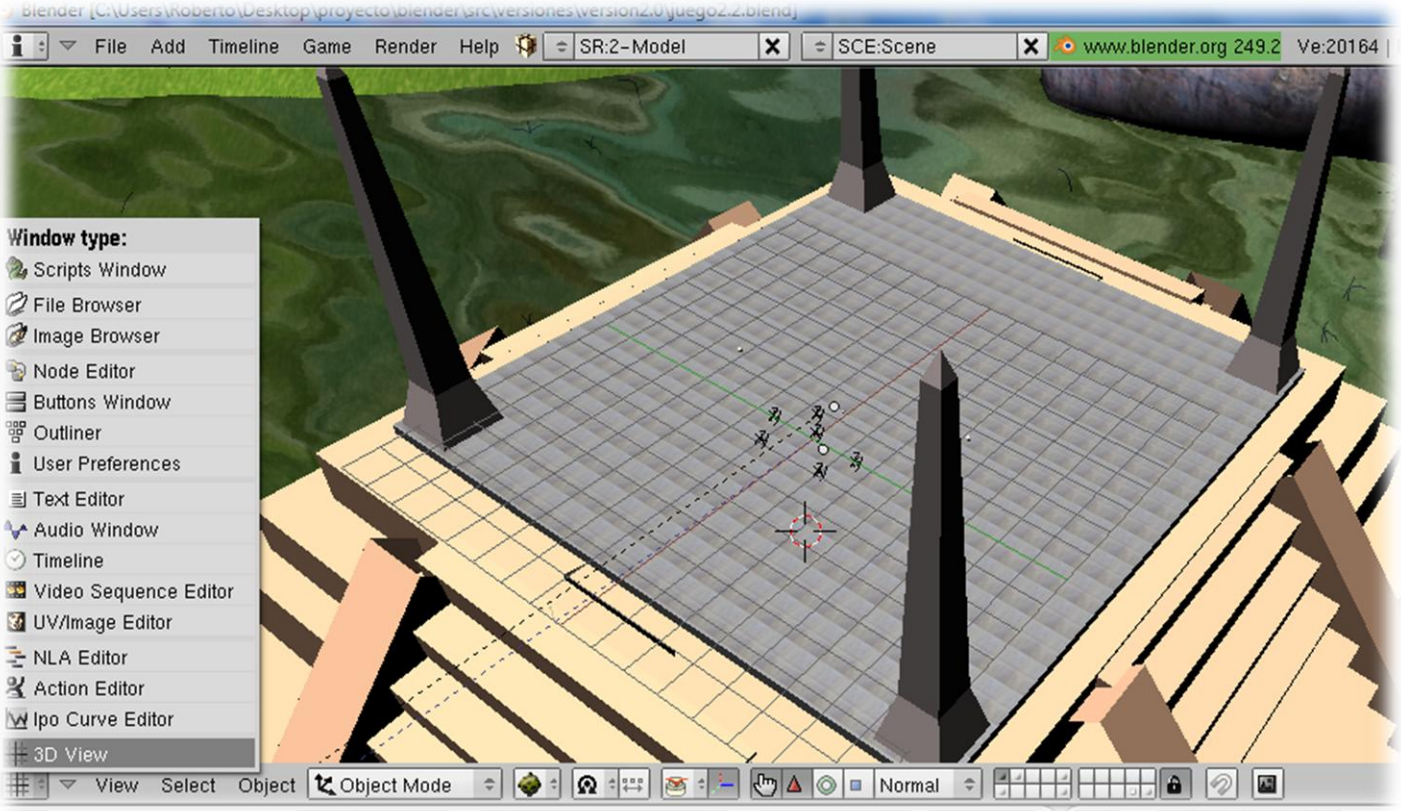


Ilustración 42. Vista 3D

Para realizar el seguimiento de este apartado del manual, se va a utilizar la ventana "*3D View*" o vista 3D que se puede observar en la ilustración 42 en el menú "*Window Type*" desplegado. En esta ventana es en la que se manipulan, crean y modifican todos los objetos visibles del juego. También es en esta ventana en la que podemos acceder a las distintas capas mencionadas en apartados anteriores.

Movimiento en el entorno 3D:

Para moverse en el espacio tridimensional, se puede hacer en dos vistas, una global y una local. Este tipo de vistas se puede cambiar en el menú "*View*", (vista), que se muestra desplegado en la ilustración 43.

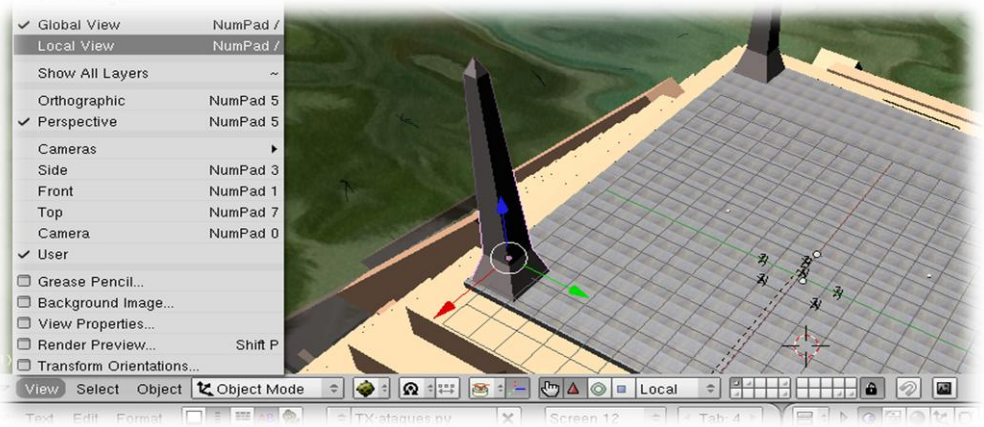


Ilustración 43. Vista local y global

Si seleccionamos una vista global, en la pantalla aparecerán todos los objetos de la escena, y nos podremos mover por la misma libremente del siguiente modo:

Si se quiere rotar la cámara que nos proporciona la visión del entorno, hay que mantener pulsada la rueda del ratón, y desplazarlo en la dirección que se desee girar. Si lo que se pretende es desplazar la cámara de modo paralelo a la vista que tenemos hay que pulsar la tecla SHIFT, mantener pulsada la rueda del ratón, y desplazarlo en la dirección que se desee desplazar la vista. Si se quiere acercar o alejar la vista, simplemente habrá que girar la rueda del ratón hacia adelante o hacia atrás.

En la vista global cualquier movimiento de la cámara, se realizará respecto del centro de la vista que se tenga en el momento del movimiento. En cambio cuando se cambie a la vista local, sólo se verá en pantalla el objeto seleccionado al realizar el cambio de objeto, y todos los movimientos tomarán como pivote el objeto en cuestión, realizándose así todas las rotaciones alrededor del mismo. Las combinaciones de teclas para realizar los movimientos en vista local son exactamente las mismas que en la vista global.

Transformaciones de los objetos en el espacio

Existen varias secciones del interfaz de la ventana "*3D View*" que sirven para transformar un objeto seleccionado. En primer lugar en la ilustración 44, se puede ver el menú que nos permite trasladar (triangulo rojo), rotar (corona verde) o escalar (cuadrado azul), un objeto.

Justo al lado de estos 3 símbolos, existe un menú desplegable que permite cambiar los ejes respecto a los que se van a realizar los movimientos. Si se escoge la orientación "*Global*", los objetos realizarán sus transformaciones respecto a los ejes globales del espacio 3D, que son fijos y no

cambian. Si se escoge la orientación "*Local*", los objetos realizarán sus transformaciones respecto a los ejes propios del objeto, que rotan y cambian con cada movimiento que sufre el objeto.

Esta orientación existe por ejemplo para saber cuál es la parte delantera de un objeto y cual la trasera, lo que es muy útil a la hora de mover a un personaje dentro de un juego. Si se escoge la orientación "*Normal*", los objetos realizarán sus transformaciones respecto a la normal del objeto seleccionado. Y por ultimo si se escoge la orientación "*View*", los ejes del objeto se alinearán con la vista que se tenga en ese momento del mismo, lo cual es útil para algunas animaciones en las que se quieren realizar movimientos realistas desde un punto de vista diagonal al objeto.

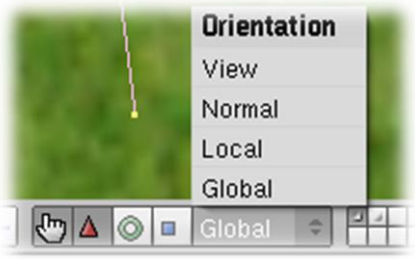


Ilustración 44. Detalle de herramientas de transformación de objetos

Otro punto a tener en cuenta a la hora de realizar una transformación sobre un objeto, es el punto respecto al cual se va a realizar dicha trasformación. Por ejemplo, cuando se quiere escalar una esfera para hacerla más pequeña, si el punto respecto al que se escala, es su propio centro, se verá como la esfera se encoge de un modo aparentemente natural sin realizar desplazamientos. En cambio, si se escala la esfera respecto a la posición del cursor, y el cursor se encuentra a la derecha de la misma, lo que ocurrirá será que la esfera se irá encogiendo hacia la derecha, desplazándose a la vez que se encoge. Al punto de referencia respecto del que se transforman los objetos se le llama pivote, y se puede cambiar con el menú emergente mostrado en la imagen inferior.

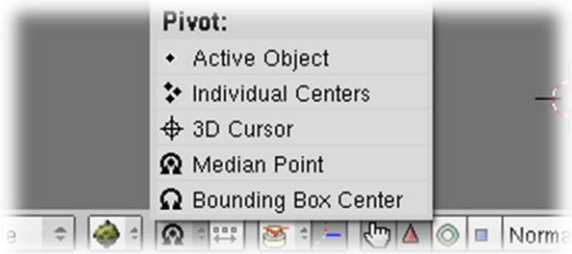


Ilustración 45. Detalle de opciones de pivote

Aunque este apartado pueda parecer algo fácil y superficial, es necesario tener estos conceptos muy claros para no cometer errores a la hora de diseñar un personaje, crear movimientos para una animación, o simplemente situar objetos en la escena. Es imprescindible conocer que partes del interfaz hacen referencia a la vista del propio usuario, y cuales hacen referencia al movimiento y transformaciones de los objetos del juego.

Resumen de menús de diseño y comandos imprescindibles para el desarrollo del juego

En este último sub-apartado se explican las funciones necesarias para añadir objetos al juego, modelarlos y editarlos, no se trata de un apartado que explique cómo diseñar un personaje realista o un paisaje bien hecho, ya que eso depende más de la imaginación y capacidad artística de cada uno, que del conocimiento que tenga del programa en sí.

Dicho esto se comenzará por explicar los modos de edición existentes, y después se explicarán los distintos menús y comandos aplicables al diseño 3d que más se utilizan.

Existen tres modos principales de edición en *Blender*, dos de ellos para modificar cualquier tipo de objeto, y uno de ellos que sólo aparece cuando se tiene seleccionado un esqueleto. Estos modos son "*Object mode*" que sirve para hacer transformaciones sobre el objeto entero, "*Edit mode*", que permite seleccionar los vértices, aristas o caras de las mallas que forman los objetos y hacer transformaciones sobre los elementos seleccionados, y "*Pose Mode*" que sirve para hacer transformaciones sobre los distintos huesos de un esqueleto y así poder crear animaciones. Estos tres modos se pueden seleccionar en el menú mostrado en la ilustración 46.

Ilustración 46. Menú de modos

Para agregar cualquier objeto de los que ofrece por defecto *Blender*, tales como polígonos regulares, curvas, superficies, texto, cámaras, luces, etc, basta con acceder al menú "*Add*" (añadir), que se muestra en la ilustración 47.

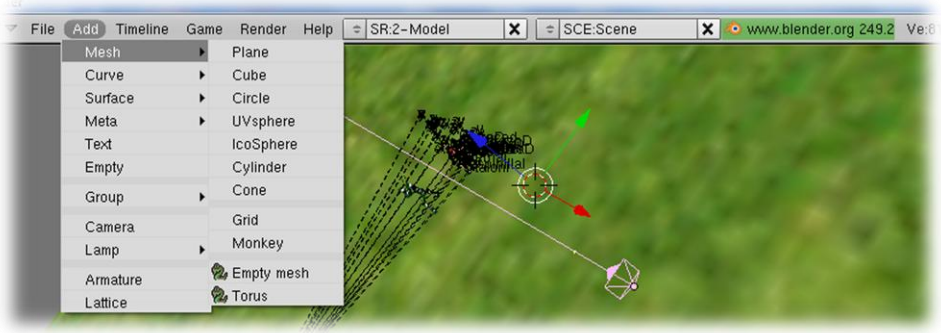


Ilustración 47. menú añadir

A continuación se muestra una tabla que contiene las funciones más utilizadas e importantes para el diseño en *Blender*, junto con una explicación de lo que hacen, y el acceso rápido desde el teclado, los cuales puedo asegurar por propia experiencia que resultan más que útiles a la hora de ahorrar tiempo en el diseño. El hecho de incluir estas funciones en una tabla, es porque esto supondrá una manera más fácil y rápida a la hora de consultar un comando en el manual, que si estuviese todo escrito en párrafos. Así se podrá buscar directamente en las diferentes columnas la información solicitada. En la siguiente tabla, se expone en la primera columna el nombre de la acción a realizar, en la segunda el acceso rápido a través de combinaciones de teclas, en la tercera, una pequeña descripción de lo que hace dicha acción, y en la última, el modo de edición en el que hay que estar para poder ejecutarla.

Funciones y comandos más útiles para el diseño 3D			
NOMBRE	**ACCESO RÁPIDO**	**DESCRIPCIÓN**	**MODO**
Escalar	s	Pulsando s, al desplazar el ratón, el objeto se encogerá si acercamos el puntero del ratón al pivote, y se agrandará si lo alejamos.	todos
Eliminar escalado	Alt+s	Elimina todos los escalados del objeto seleccionado	todos
Rotar	r	Pulsando r, el objeto rotará en el sentido que movamos el ratón.	todos
Eliminar rotación	Alt+r	Elimina todas las rotaciones del objeto seleccionado	todos
trasladar	g	Pulsando g, el objeto se trasladará en el sentido que traslademos el ratón.	todos
Eliminar traslación	Alt+g	Elimina todas las traslaciones del objeto seleccionado.	todos
Duplicar	Shift+d	Duplica el objeto seleccionado, incluyendo transformaciones, lógica, atributos, materiales y todo en general. Si estamos en "*Edit Mode*", podremos seleccionar vértices, aristas, o caras y duplicarlas del mismo modo.	Edit mode Object mode
Extrudir	e	Extrudir es una de las herramientas más útiles del diseño, y se basa en la extensión de una cara, arista o vértice o de todo un polígono, de modo que ese elemento se duplique y se extienda, pero manteniéndose unido al original por medio de aristas. El elemento que normalmente se extrude es una cara, por ejemplo de un cubo, de modo que si se extrude la cara derecha del cubo moviendo el ratón hacia la derecha, la figura resultante podrían ser dos cubos pegados, o si se alargase más el nuevo cubo, podría ser un cubo con un prisma rectangular pegado a su derecha. Si lo que se	Edit mode

		extrude en cambio es el cubo entero, la figura obtenida será un cubo con un cubo adicional pegado a cada una de sus caras.	
emparentar	Ctrl + p	Seleccionando un objeto, después pulsando Ctrl+p y seleccionando otro objeto con el botón derecho del ratón, se consigue que el segundo objeto sea padre del primero. Un objeto hijo sufre las mismas transformaciones que se aplican a un objeto padre, de modo que si creamos una persona por un lado, y una capa por ejemplo por otro lado, y emparentamos la capa al cuello del cuerpo, si movemos a la persona en cualquier dirección, la capa la seguirá manteniéndose siempre en la misma posición relativa al cuello de esa persona.	todos
desemparentar	Alt + p	Elimina el parentesco del objeto seleccionado.	todos
cortar	k	Pulsando K podemos cortar la arista o polígono que se tenga seleccionado en modo de edición con 4 opciones de corte distintas, corte en bucle(cortará el objeto empezando y acabando por la arista seleccionada, creando un círculo o polígono cerrado alrededor del mismo), corte a la mitad (crea un corte en los puntos medios de las aristas seleccionadas), corte exacto (cortará las arista por donde se vaya moviendo el puntero del ratón) o multi-corte (crea tantos cortes como se le indique en un parámetro numérico). Tras señalar los cortes con el ratón, se deberá pulsar "*enter*" para confirmar la operación.	Edit mode
subdividir	w	Pulsando w aparece un menú de funciones especiales entre las que se encuentra subdividir, en 4 modalidades diferentes. Subdivide (divide en 2 el polígono o cara seleccionados), Subdivide Multi (divide el polígono o cara seleccionados de forma regular en el número de divisiones indicadas por el parámetro "*Number of Cuts*"), Subdivide Fractal (divide el polígono o cara seleccionados de forma fractal en el número de divisiones indicadas por el parámetro "*Number of Cuts*").	Edit mode
Confluir	Alt+m	Confluir ("*Merge*" en el interfaz es unir los vértices seleccionados en uno solo)	Edit Mode
Espejo	Ctrl+m	Crea un objeto especular al objeto seleccionado, a lo largo de los ejes x y o z. Se trata de una función muy útil a la hora de crear objetos simétricos, ya que sólo habrá que modelar la mitad, y crear la otra mitad como si de un espejo se tratase.	Object Mode
Hacer local	l	Convierte un objeto que no sea local (importado o enlazado desde otro archivo), en un objeto local.	Object Mode
Mover a	m	Mueve el objeto seleccionado a la capa que se elija	Object Mode

capa		al pulsar el la tecla m.	
Panel numérico	n	Pulsando la tecla n aparece el panel numérico con los datos numéricos de escalado traslación y rotación del objeto seleccionado, y permite modificarlos para lograr transformaciones más precisas.	todos
deshacer	Ctrl+z	Deshace la última operación sea cual sea.	todos
combar	Shift+w	Comba los vértices seleccionados.	Edit Mode
Objeto especular	m	Similar a la función espejo, pero esta no duplica el objeto si no que lo elimina y crea su imagen especular.	Edit Mode
borrar	x	Borra el objeto seleccionado.	todos
Añadir objeto	Barra espaciadora	Abre el menú "*Add*" comentado anteriormente en este mismo apartado	todos
Vista frontal	1	Vista perpendicular al plano xz	todos
Vista lateral	3	Vista perpendicular al plano yz	todos
Vista superior	7	Vista perpendicular al plano xy	todos
Vista de cámara	0	Vista de la cámara activa. En esta vista veremos lo que posteriormente e verá en el juego compilado	todos
Grupos	Ctrl+g	Este comando nos permite crear grupos de objetos, sacar un objeto seleccionado de un grupo, o añadir un objeto seleccionado a un grupo ya existente.	Object Mode

Tabla 4. Funciones y comandos más útiles para el diseño 3D

4.2.2. Creación de un esqueleto y fijación del mismo a una malla

Creación de un esqueleto

Para empezar a crear un esqueleto se debe presionar la barra espaciadora en la que aparecerá la opción para añadir un hueso, "*Armature*" en el interfaz. Como se puede observar en la ilustración 48, al presionar esta opción del menú, aparecerá un hueso, que es el objeto con forma de pirámide que aparece perfilado en rosa en la ilustración 48.

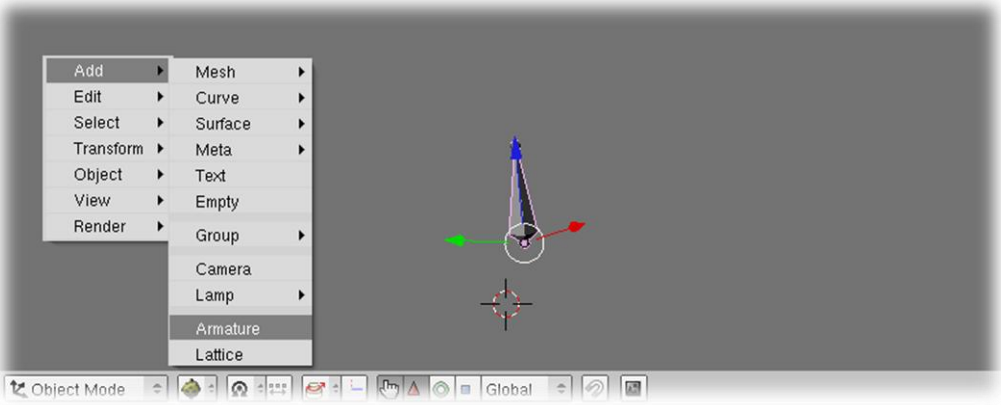
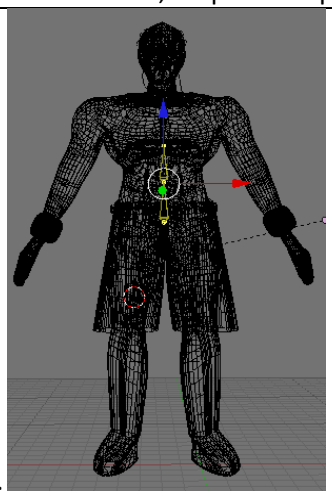
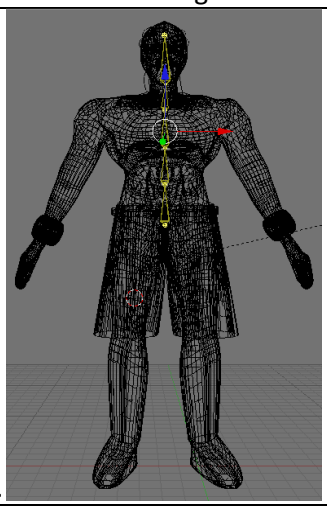
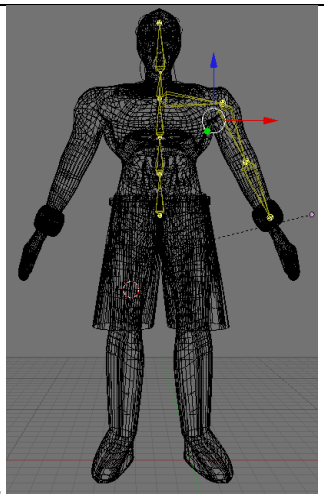
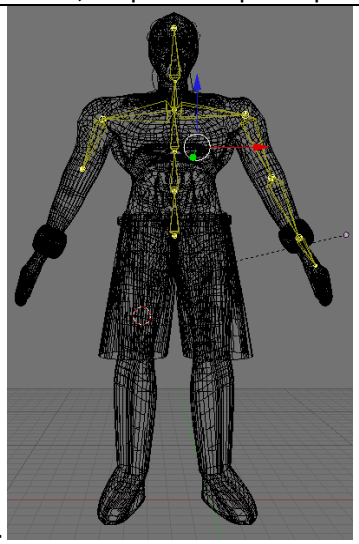
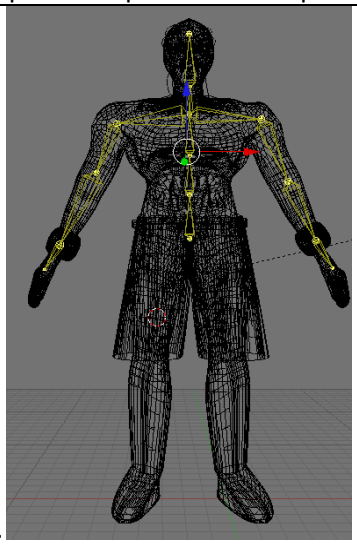
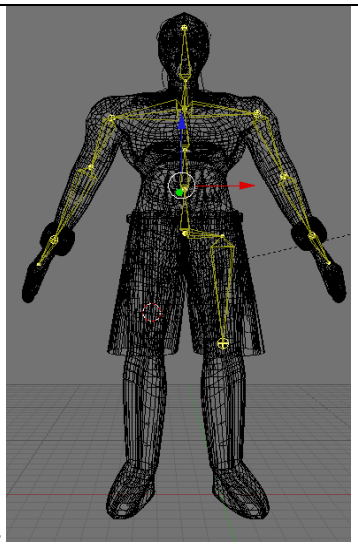


Ilustración 48. Inserción de huesos

Un esqueleto está formado por varios de estos huesos que se van añadiendo encadenados unos a otros, creándose una relación de parentesco para cada hueso, con el hueso anterior. Para añadir huesos encadenados a partir del primero, es necesario entrar en el modo de edición, "*Edit Mode*", seleccionar el extremo fino del hueso, pulsar la tecla Ctrl, y con ella pulsada, pulsar el botón izquierdo del ratón en el punto del espacio donde se desea que acabe el nuevo hueso.

Para crear un esqueleto que se ajuste bien a la malla que tiene que mover, es conveniente crear el esqueleto sobre dicha malla, utilizándola a modo de guía. Si se quiere que un personaje u objeto se mueva de un modo realista, ente cada articulación que tenga dicho objeto deberá existir un hueso. La unión entre dos huesos será la articulación que se podrá doblar como si de una real se tratase. Mediante la siguiente secuencia de imágenes, se pretende recrear paso a paso el proceso de creación de un esqueleto para una malla que representa un ser humano.

<table>
<tr><td colspan="3">Creación de la columna desde la cintura hasta la cabeza, y posterior creación del brazo derecho desde el cuello, empezando por la clavícula hasta llegar a la mano.</td></tr>
<tr><td>1.
  </td><td>2.
  </td><td>3.
  </td></tr>
<tr><td colspan="3">Creación del brazo izquierdo de nuevo desde el cuello y posterior creación del fémur de la pierna derecha, empezando por la pelvis, a partir del primer hueso que había sido creado.</td></tr>
<tr><td>4.
  </td><td>5.
  </td><td>6.
  </td></tr>
</table>

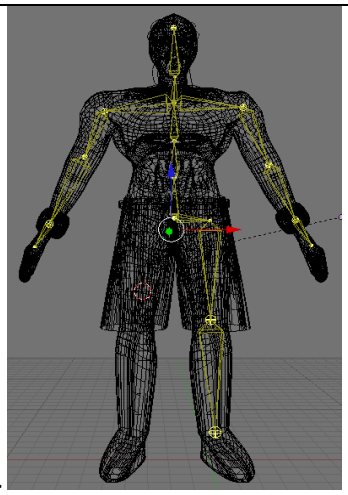
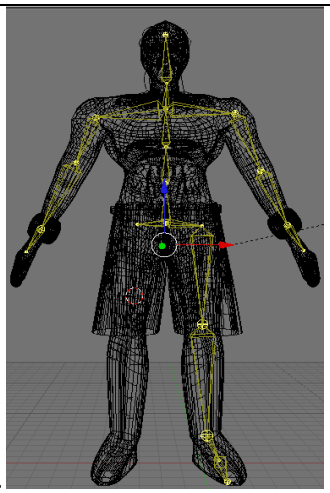
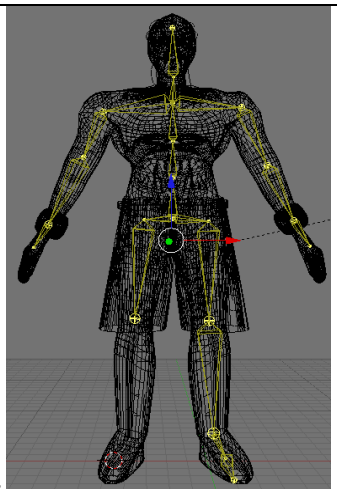
Finalización de la pierna y pie derecho y comienzo con el fémur de la pierna izquierda.		
7. 	8. 	9. 

Tabla 5. Secuencia de creación de un esqueleto

Como se puede apreciar en la secuencia anterior se ha empezado por el hueso situado en la cintura del personaje. Esto es así porque las personas realizan los movimientos de sus extremidades de forma relativa a la cintura, de modo que si suben una pierna, la cintura no se mueve, o si doblan el torso la cintura tampoco se desplaza de lugar.

En cambio cuando queremos andar, saltar, o cualquier movimiento que requiera un desplazamiento, la cintura también se desplaza y todas las demás partes del cuerpo la siguen, moviéndose de modo independiente pero siempre con un movimiento relativo a la posición de la cintura.

El hecho de empezar por el hueso de la cintura significa que dicho hueso va a ser el padre de todos los que estén conectados a él, y todos esos huesos tendrán a la vez sus hijos, y así sucesivamente hasta llegar a los extremos del esqueleto.

Los huesos hijos, tienen la propiedad de realizar las mismas transformaciones que se ejecutan sobre los padres de una manera proporcional, de este modo si queremos desplazar el esqueleto entero, bastará con desplazar el hueso inicial. Qué ocurre si además se quiere levantar la pierna del personaje. Para realizar esta acción sólo es necesario rotar el fémur hacia arriba, y todos los huesos hijos rotarán respecto del mismo pivote manteniendo así la unión entre los mismos.

Sabiendo esto, a la hora de diseñar un esqueleto, no sólo habrá que tener en cuenta la anatomía o estructura del objeto al que se le va a asignar, sino que también habrá que conocer como se mueve para decidir que huesos van a ser padres y cuales hijos, y cuantos huesos deben formar dicho esqueleto para que los movimientos generados posteriormente sean lo más realistas posibles.

Por último es conveniente conocer la existencia de un menú que nos permite cambiar alguna de las propiedades de los huesos, como sus nombres, la forma en la que se ven representados en la pantalla, o el modo en que afectarán a la malla una vez estén asignados a la misma. Este menú está en la ventana "*Buttons window*", en el subapartado "*Editting*". Será útil dar nombres descriptivos a los huesos ya que esto nos permitirá realizar la operación de "*skinning*" o "*rigging*" de una manera más sencilla.

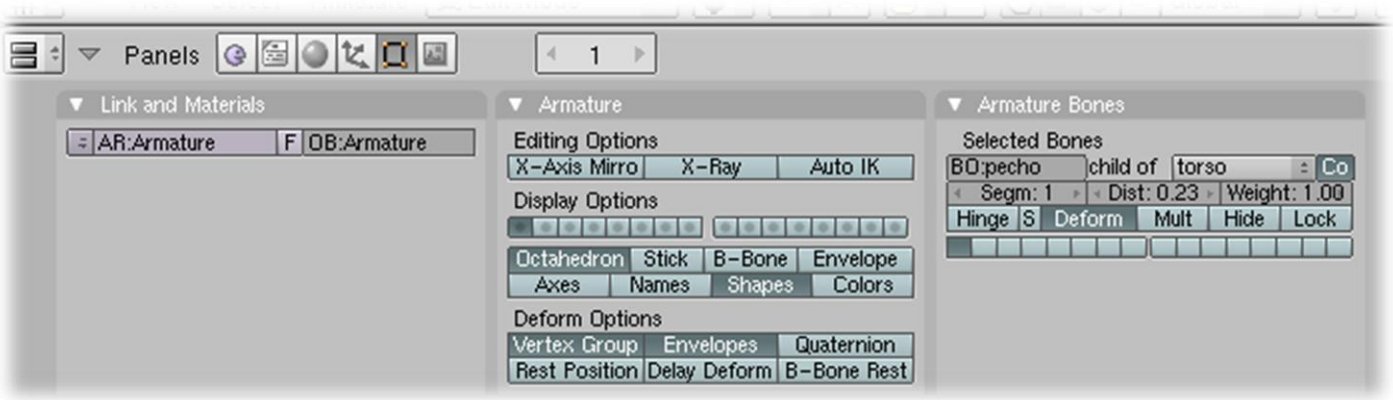


Ilustración 49. Menú de edición de huesos

El primer panel empezando por la izquierda es común a todos los objetos, y permite cambiar el nombre del esqueleto en su totalidad. En el caso de la imagen superior, el objeto esqueleto se llama "*Armature*".

El segundo panel permite cambiar opciones que hacen referencia a todo el esqueleto, como por ejemplo la opción de permitir que el esqueleto se vea por encima de la malla ("*X-Ray*"), el tipo de representación que se quiere que tengan los huesos en la ventana 3D, o las opciones de deformación que tendrán distintos efectos sobre la malla a la que se fije el esqueleto.

También permite escribir los nombres al lado de cada uno de sus huesos para reconocerlos mejor, y poner al esqueleto en una posición de descanso, ("*Rest Position*"), muy útil a la hora de editar algún retoque tanto en la malla como en el propio esqueleto.

El último panel permite modificar opciones del hueso seleccionado, como su nombre su hueso padre, su tamaño, su peso, e incluso si se quiere que ese hueso en concreto deforme la malla o no.

Fijación del esqueleto a la malla

Para que un esqueleto consiga que una malla se mueva, hay que fijar el esqueleto a dicha malla. Esta operación se conoce como "*skinning*", y hay dos formas principales de hacerlo. Se puede emparentar un objeto a un único hueso, en este caso se trataría de una relación de parentesco normal y corriente entre dos objetos. También se puede emparentar el objeto con un esqueleto entero, de modo que a cada hueso se le asignen ciertos grupos de vértices y el esqueleto al moverse deforme la malla en función de sus movimientos.

En todos los programas modernos de diseño 3D se puede realizar esta operación de un modo más o menos estándar, pero *Blender* ofrece varias opciones que pueden ser útiles dependiendo de la situación. Estas opciones aparecen en el siguiente menú cuando se emparenta ("*Ctrl+P*") un objeto con un esqueleto completo.

Ilustración 50. Tipos de Skinning

1. Opción 1, "*Don´t create groups*":

Esta opción aparece cuando la malla a la que vamos a fijar el esqueleto no tiene grupos de vértices definidos, y lo que hace es fijar la malla al esqueleto de un modo automático sin crear grupos. Para mallas complejas esta opción no suele funcionar muy bien.

2. Opción 2,"*Name Groups*":

Crea grupos de vértices vacíos cuyos nombres concuerdan con los nombres de los huesos, pero sin asignarles vértices, es decir que se crean tantos grupos de vértices como huesos tiene el esqueleto, pero ningún vértice pertenece a dichos grupos, habría que asignárselos posteriormente, volviendo a emparentar el esqueleto a la malla.

Estas dos primeras opciones no son demasiado utilizadas, la primera porque ha sido mejorada con las opciones 3 y 4, y la segunda porque el proceso de asignación manual de vértices es un proceso muy pesado, y porque las otras tres opciones existentes lo hacen de modo automático.

3. Opción 3, "*Create From Envelopes*":

Utiliza envoltorios para conocer las zonas de influencia de los huesos sobre la malla. Los envoltorios son áreas de influencia que rodean a cada hueso y que se pueden modificar. Cualquier vértice de una malla que este dentro de un envoltorio, será controlado por el hueso que genera dicho envoltorio. De este modo se tiene una fijación que puede cambiar con sólo cambiar la forma o tamaño de los envoltorios.

Para poder ver los envoltorios de un esqueleto, es necesario seleccionar el esqueleto, entrar en el modo de edición, acceder al menú de edición de huesos explicado con anterioridad, y pulsar el botón "*envelopes*". De este modo obtendremos algo parecido a la ilustración 51, en la que los huesos aparecen mostrados en gris, y alrededor del hueso seleccionado en rosa, se puede ver el envoltorio blanco trasparente que rodea la cabeza.

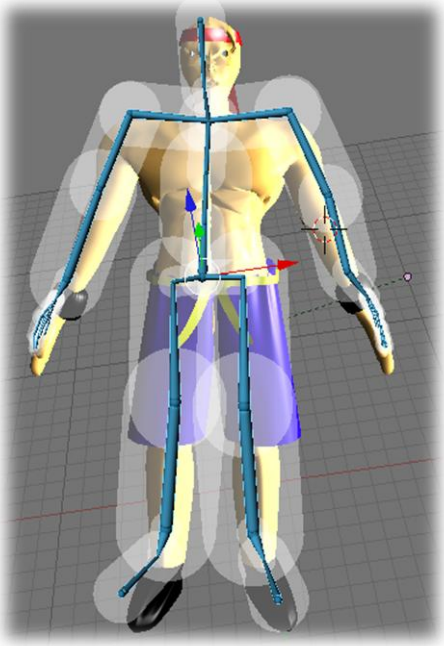


Ilustración 51. Envoltorios de un esqueleto

Para cambiar el tamaño del envoltorio, existen 2 métodos:

El primero seleccionando el extremo de un hueso y escalándolo pulsando la tecla "s", de modo que dicho extremo del envoltorio se escalará a la vez que lo hace el extremo del hueso.

El segundo método se basa en seleccionar un hueso entero y escalar solamente el envoltorio pulsando "Alt+s".

4. Opción 4, "*Create From Bone Heat*":

Esta opción suele dar mejores resultados en la mayoría de las ocasiones, pero como ya se ha comentado con anterioridad, todo depende de la estructura del esqueleto y de la malla que se esté

tratando. Este método, asigna grupos de vértices a los huesos que estén situados más cerca de dichos vértices, pudiendo estar asignado un único vértice a varios huesos.

Si un vértice está asignado a varios huesos, la influencia de estos huesos sobre dicho vértice dependerá de la distancia del punto a cada uno de ellos, siendo mayor cuanto más cerca del hueso este situado.

5. Refinamiento del proceso de fijación:

Una vez se ha hecho el "*skinning*", con alguno de los métodos anteriores, se observará que en la malla se han creado grupos de vértices, con los nombres que tienen cada uno de los huesos a los que están asignados. Es posible que se vean vértices, que por estar muy alejados, no estén asignados a ningún hueso, y por tanto no se muevan junto con el resto del esqueleto, o vértices que se hayan asignado a huesos que no son los correctos y produzcan deformaciones extrañas al manipularlos.

Para arreglar estos problemas se accederá en la ventana "*Buttons window*", en Modo de edición y después a la sub-sección de edición que se puede observar en la ilustración 52. En este menú se pueden manipular los grupos de vértices de la malla seleccionada. Pueden manipularse de los siguientes modos: eliminándolos del grupo al que pertenecen, añadiéndolos a otros grupos existentes o creando nuevos grupos de vértices.

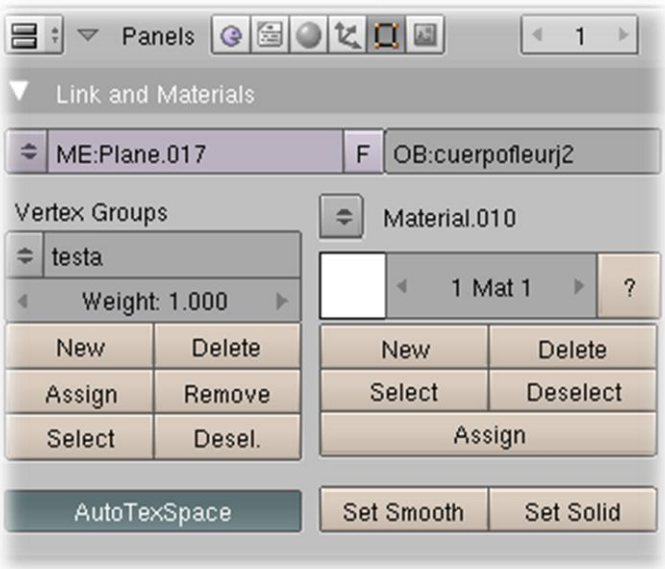


Ilustración 52. Asignación manual de vértices

En la imagen superior se puede observar el grupo testa seleccionado en el menú desplegable. Si se pulsa el botón "*select*", se seleccionarán los vértices pertenecientes a dicho grupo, en la ventana 3D, en este caso la cabeza.

Si por ejemplo un vértice del cuello se hubiese asignado a la cabeza por error, habría que seleccionar dicho vértice en la ventana 3D, después habría que seleccionar en el menú desplegable, el grupo de vértices "*testa*" o cabeza, y pulsar el botón "*remove*". De este modo se elimina el vértice

del grupo de vértices de la cabeza. Para añadir este vértice al cuello, se debería seleccionar en el menú desplegable el grupo de vértices del cuello, y con el vértice seleccionado en la ventana 3D, presionar el botón "*Assign*".

4.2.3. Texturas y materiales

Materiales

En *Blender* se le puede asignar a un objeto uno o varios materiales, y cada material puede estar compuesto por una o más texturas. Los materiales definen las propiedades físicas de un objeto, como puede ser la luz que absorbe, la que refleja, su transparencia, color base, rugosidad, etc. Las texturas en cambio definen el dibujo que se va a ver en la superficie de dicho material.

Para empezar a trabajar con materiales, lo primero que hay que hacer es acceder al menú de materiales, que se encuentra en la ventana "*buttons window*", en la sub sección "*Material buttons*".

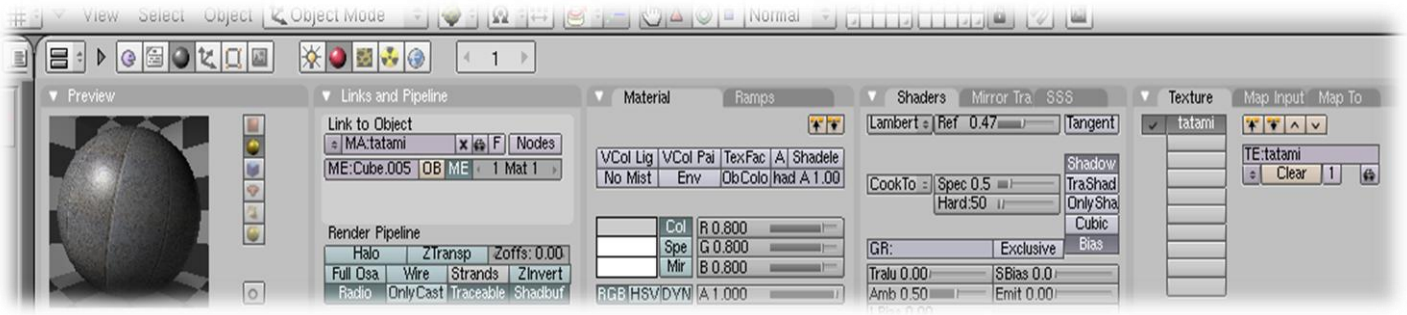


Ilustración 53. Menú de materiales

Como se puede observar en la ilustración 53, la primera pestaña del interfaz ("*Preview*"), muestra una pre-visualización del material editado en tiempo real.

La segunda pestaña, ("*Links and pipeline*"), permite modificar el nombre del material en cuestión mediante el campo "*MA*:", además de elegir cualquier otro material de los disponibles en el archivo, para el objeto que seleccionado en ese momento. En el apartado "*Render Pipeline*", se permiten seleccionar algunas opciones para el renderizado de materiales, como pueden ser la transparencia, sombreado o el que sólo se muestren los hilos de la malla.

Las dos siguientes pestañas son parámetros a partir de los cuales podremos obtener distintos efectos para los distintos materiales, y que no merecen mayor mención en este manual. En cambio la ultima pestaña, que está formada por otras tres pestañas a su vez, si es importante y conveniente conocerla a fondo, ya que es indispensable para poder añadir texturas a los objetos de un videojuego.

Texturas

Aunque existe una sub sección ("*texture*"), al lado de la sub sección de materiales, en esa sección sólo se elige como va a ser el dibujo que va a tener estampada una textura en su superficie según diferentes parámetros, como el tipo de textura, el uso de canales alfa, repeticiones, etc.

Ilustración 54. Edición de texturas

Una vez creada una textura, se utilizarán los tres paneles mostrados en la ilustración 55, para decidir cómo se va a utilizar dicha textura sobre el material al que se le ha asignado.

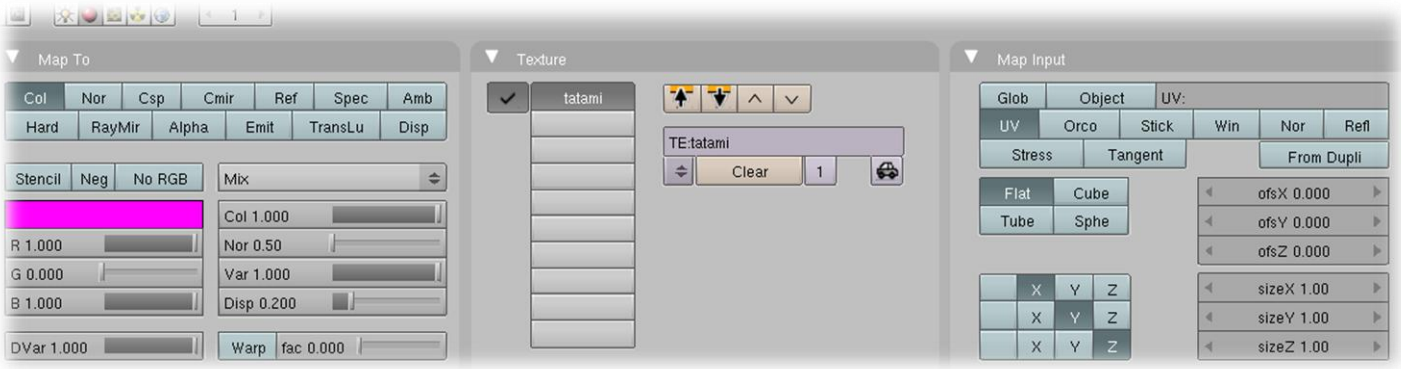


Ilustración 55. Texturas dentro de un material

El primer panel de la izquierda se utiliza para indicar al material sobre cuál de sus atributos se va a mapear la textura seleccionada, se puede mapear directamente sobre el color, es decir, que se plasma el dibujo de la textura sobre la superficie del material, pero por ejemplo, también se puede mapear sobre la normal de la superficie, creando "*bumps*" o irregularidades sobre la superficie del objeto, cuya profundidad depende de la intensidad de los colores de la textura. De este modo se pueden afectar los valores de dureza, especular, canal alfa, y en definitiva todos los que aparecen en la ilustración 55. Al pasar el ratón por encima de ellos se obtiene una explicación del valor al que afectara la textura si se presiona dicho botón.

El segundo panel sirve para indicar que texturas están activas y cuáles no, y que textura se está editando al modificar parámetros en los otros dos paneles.

En el tercer panel, se indica cómo se va aplicar la textura a la superficie 3D, en términos de coordenadas. Aquí se decide si la textura se va a estirar hasta que ocupe toda la superficie, si se va a repetir hasta cubrir toda la superficie, o también se pueden fijar estas coordenadas manualmente mediante el mapeado UV, el más importante y más utilizado para mapear las texturas de prácticamente todos los objetos de un videojuego.

Si por ejemplo se quiere crear un material que simule agua, serán necesarias dos texturas. La primera textura será azul, transparente, y estará mapeada sobre el parámetro "*Col*" del panel "*Map To*". Esta configuración dará el color del agua al plano utilizado. Para conseguir el oleaje típico de un lago por ejemplo, será necesario añadir otra textura al material que afecte a los valores de la normal del plano, produciendo así deformaciones que simulen olas en el agua.

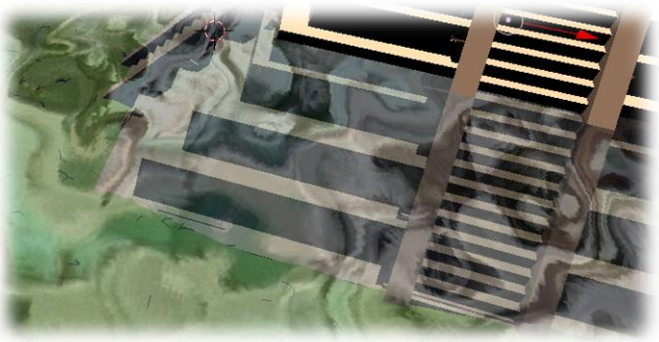


Ilustración 56. Ejemplo de textura de agua

Mapeado UV

Puesto que el mapeado UV [37] es muy utilizado en los videojuegos se le ha dedicado un apartado en el que se explica cómo utilizar esta técnica, y como aplicarla en *Blender*, a partir de un ejemplo de mapeado para un personaje humano.

La manera típica de crear una textura para un personaje, es dibujarla, y después asignársela entera al cuerpo del personaje. En *Blender* existe una ventana dedicada exclusivamente a esta tarea. Esta ventana se llama "*UV/Image Editor*" y se muestra en la imagen inferior. El dibujo de la imagen puede parecer un cuerpo deformado, pero está hecho así a propósito para que encaje bien a la hora de proyectar la maya sobre la textura en 3 dimensiones.

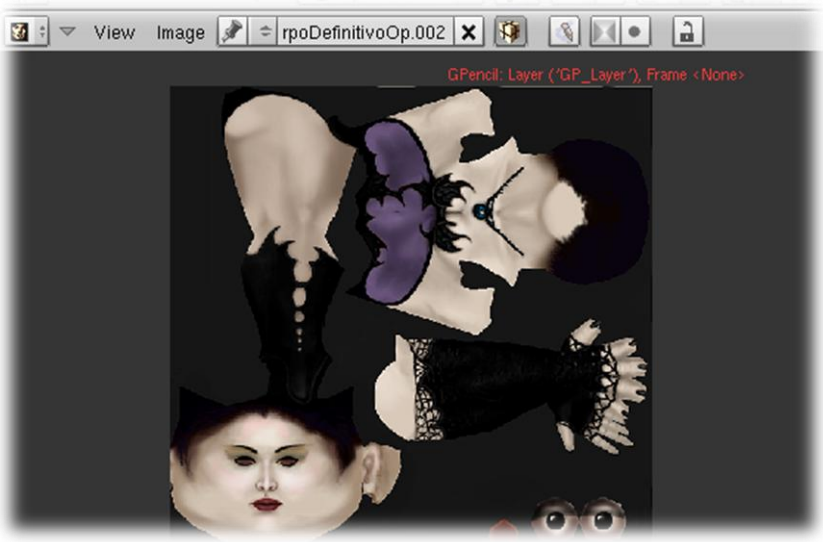


Ilustración 57. Ejemplo de textura de cuerpo completo

Esta ventana permite cargar una imagen en *Blender* y proyectar sobre ella la malla del objeto al cual se la queramos asignar. Además permite transformar los vértices proyectados sobre la imagen a nuestro antojo, para conseguir un mejor ajuste de la textura a la malla. Los comandos para transformar los vértices en la ventana "*UV/Image Editor*" son los mismos que los utilizados en la ventana "*3D View*".

Para proyectar la malla sobre la imagen elegida, hay que seleccionarla en "*Object Mode*", pasar a "*Edit Mode*", pulsar la tecla "A" para seleccionar todos los vértices de la malla, y pulsar la tecla "U", que nos mostrará el menú "UV calculation" desplegado en la ilustración 58. En este menú se deberá pulsar en "Unwrap", y se podrá ver como la malla aparece desenvuelta sobre la imagen abierta en la ventana"UV/Image Editor", y como en la ventana 3D aparece la textura sobre la superficie del personaje. El resto de opciones del menú son distintas formas de proyectar la maya sobre la imagen.

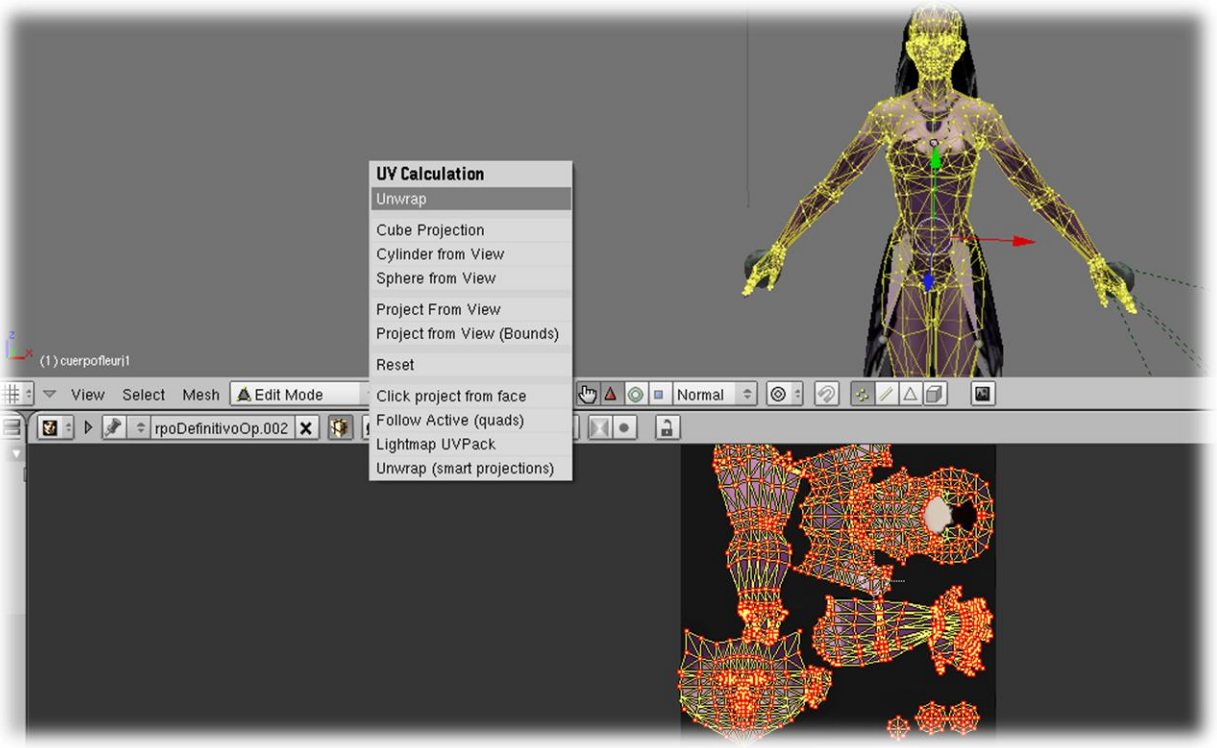


Ilustración 58. Ejemplo de texturas UV

Una vez hecho esto, el siguiente paso es acceder al menú de materiales, crear una nueva textura de tipo imagen, cargar la misma imagen utilizada hasta ahora en el menú de creación de texturas, y de vuelta en el menú de materiales, en el panel "Map input", seleccionar la opción "UV". De este modo quedará creada una textura con la imagen seleccionada, y el mapeo creado manualmente en la ventana"UV/Image Editor" sobre dicha imagen.

Si queremos utilizar diferentes imágenes para una misma malla, bastará con desenvolver sólo la parte de la malla a la que se desea asignar la imagen como textura.

4.3.IPO y acciones

IPO y Acción en *Blender*, hacen referencia a una misma función que se puede realizar de dos modos diferentes. Las IPOS se suelen usar para transformaciones globales de un objeto, y las acciones almacenan las animaciones creadas a partir de un esqueleto, o de una IPO ya creada, que posteriormente se puede convertir en Acción.

4.3.1. IPO

Las "*IPO*" se utilizan por ejemplo, cuando se quiere hacer crecer un objeto, desplazarlo cuando ocurre algún tipo de evento, realizar alguna acción repetitiva, como un bloque que se mueve de una pared a otra, y en general cualquier transformación que afecte a todo el objeto, y no sólo a una parte del mismo.

En el videojuego implementado para la realización de este manual, se utiliza una IPO, para crear el efecto de una bola de energía que crece, se aplasta y gira al mismo tiempo. Esta animación se muestra a continuación.

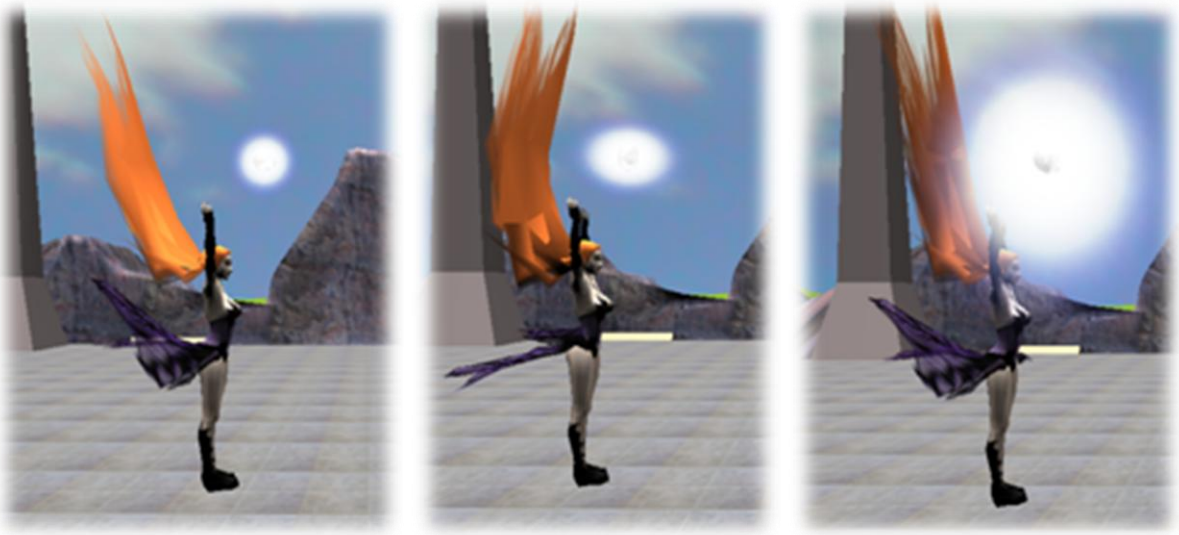


Ilustración 59. Secuencia IPO bola energía

Para Crear una "*IPO*" es necesario seguir los siguientes pasos:

1- Lo primero que hay que hacer es acceder a la ventana "IPO curve editor".
2- Seleccionar el objeto para el que se quiere crear la IPO en el menú desplegable que hay en la parte superior del interfaz.
3- Escribir el nombre que se quiera dar al la IPO.
4- Seleccionar el *"frame"* en el que se quiere que comience la animación, típicamente el *"frame"* 1.
5- Desplazarlo, rotarlo, y/o escalarlo hasta obtener la transformación inicial deseada.
6- Pulsar la tecla "i", en el menú emergente que aparece y seleccionar el tipo de transformación que queremos almacenar "*Loc, Rot, Scale*".
7- Seleccionar el *"frame"* final de la animación, dependiendo de lo que se desee dure la misma, y repetir los pasos anteriores para almacenar la transformación final.

Con esto se consigue que *Blender* genere los pasos intermedios de la animación. Una vez creada la animación del objeto, se podrán desplazar, escalar y duplicar en la línea temporal las distintas claves almacenadas, seleccionando la curva IPO adecuada, con los mismos comandos que se transforma un objeto en la ventana 3D.

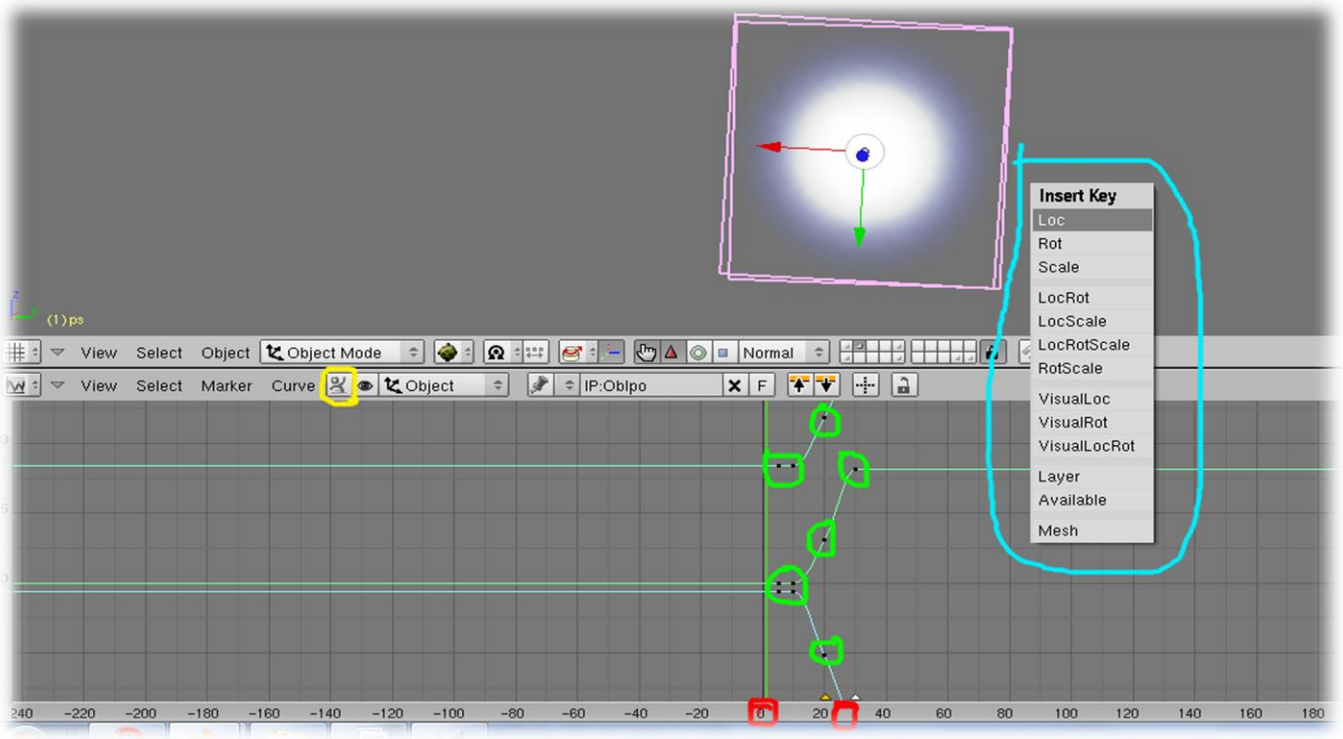


Ilustración 60. Ejemplo interfaz IPO

En la ilustración 60 se puede ver un ejemplo de IPO creada. Los números rodeados en rojo son los *"frames"* que dura la animación, los puntos negros rodeados en verde, representan los *"frames"* en los que se ha almacenado una posición fija. El resto de *"frames"* intermedios, son generados automáticamente por *Blender*.

Rodeado en azul claro, se puede ver el menú que aparece al presionar la tecla "i", y que permite elegir qué tipo de transformación se desea memorizar en el *"frame"* actual. Rodeado en amarillo está el botón que nos permitirá guardar la IPO como una acción, y por último a su derecha, con el campo "*IP:Obipo"*, está el menú desplegable que nos va a permitir cambiar de nombre a la IPO, crear una nueva, o seleccionar una de las ya existentes.

4.3.2. Acciones

Las acciones funcionan de un modo casi idéntico a como lo hacen las IPOS, salvo que las transformaciones que se almacenan son las transformaciones de cada hueso de un esqueleto. Los pasos a seguir para crear una acción son exactamente los mismos que para crear una IPO, pero hay que estar situados en modo "*Pose Mode*", y en la ventana "*Action editor*".

En el juego creado para el aprendizaje y posterior realización del manual, se han creado 25 animaciones que representan el movimiento, ataques y recepción de estos ataques, para el personaje humano que se ha utilizado a modo de luchador. Esto supone un trabajo bastante costoso, pero se ha considerado necesario, para obtener un resultado aceptable para su presentación.

Hay que tener en cuenta algunas cosas a la hora de crear una acción. Los huesos se pueden escalar, rotar y desplazar por separado, pero lo más normal es rotar los huesos hasta obtener la posición deseada, o escalarlos, si se pretende hacer más grande o más pequeña alguna de las partes del cuerpo del personaje en cuestión.

Los desplazamientos que haría una persona por ejemplo al andar, no se deben incluir en la animación, es decir que para hacer que una persona ande, en la acción sólo estará recogido el ciclo de movimiento desde que avanza un pie, hasta que ese pie vuelve a su estado original, pero el personaje en sí no avanza, es como si anduviese en el sitio. El desplazamiento hacia adelante se le agregará después por separado, desplazando todo el esqueleto hacia adelante y ajustando ese desplazamiento a la velocidad de movimiento de los pies. Hay en cambio acciones que si pueden incluir un desplazamiento de todo el esqueleto, como puede ser un salto, que asciende y vuelve a descender hasta volver a su posición inicial.

El hecho de que el desplazamiento de las acciones deba empezar y terminar en el mismo punto, es debido al sistema de posiciones que utiliza *Blender*. En *Blender* todo objeto tiene unas coordenadas, que se pueden modificar con los actuadores de tipo "*Motion*", movimiento. Los actuadores de tipo "*Action*", acción, pese a que desplazan el esqueleto, no modifican el valor de esas coordenadas, Por lo que si una acción desplazase hacia delante el esqueleto, el atributo posición del esqueleto quedaría sin modificar. En este caso, el esqueleto sí que se desplazaría visualmente hablando, pero después, cuando una segunda acción fuese ejecutada, tomaría como origen de la acción, la posición que tiene el objeto guardada como suya, y se vería como el objeto da un salto hacia atrás en un solo *"frame"* y después realizaría la segunda acción.

En la siguiente página se muestra un ejemplo de secuencia, para una animación de salto creada por el autor del proyecto. En la secuencia se pueden apreciar las rotaciones realizadas para los huesos, y detalles como el movimiento del pelo y el vestido hacia arriba mientras el personaje está cayendo.

Ilustración 61. Secuencia de animación de salto

4.4.Motor de físicas de *Blender*

Blender dispone de un motor de físicas muy potente, sobre todo para renderizar videos, pero algunas de sus capacidades no resultan útiles en el desarrollo de videojuegos, debido a la gran cantidad de recursos que consume.

En muchos casos, usar las capacidades físicas de *Blender* en el motor de juego, da lugar a un nivel de eficiencia muy bajo debido al gran número de cálculos que tiene que realizar sobre cada polígono para simular sus propiedades físicas. Es por esta razón que en este apartado se explicará sólo la parte de físicas aplicable a los videojuegos.

4.4.1. Objetos estáticos y dinámicos

En *Blender* cualquier objeto puede convertirse en un objeto estático o dinámico. Los objetos dinámicos son aquellos que están sujetos a las propiedades de la física, tales como la gravedad, aplicación de fuerzas lineales o giratorias, además de la capacidad de detectar y producir colisiones. En cambio un objeto estático es todo lo contrario a uno dinámico, y carece de actuadores que permitan aplicar una fuerza sobre él.

En un juego son necesarios ambos tipos de objetos, y dependiendo del tipo de juego, se usarán objetos dinámicos, estáticos, o ambos. Por ejemplo, para un juego en el que haya un personaje humano, si se decide que su esqueleto, que es el que mueve la malla, sea dinámico, la estabilidad del modelo se vería comprometida, y no haría más que caerse para un lado u otro. En cambio para un juego de coches por ejemplo, no habría ningún tipo de problema de ese tipo ya que un coche colocado sobre 4 ruedas, con más superficie horizontal que vertical, es más difícil que vuelque.

Los escenarios son un buen ejemplo de objetos que deben ser estáticos, al menos la parte básica de los escenarios, ya que no deben estar moviéndose en el espacio, y los personajes no deben poder desplazar o mover por ejemplo una colina o un río. Aun así, sí que se pueden añadir elementos de escenografía dinámicos que queremos que sean modificados o transformados de algún modo.

Por ejemplo, se puede querer hacer arboles con hojas a las que se quiere aplicar un vórtice de viento para que las mueva, paredes que se quiere que se rompan al golpearlas, bloques que hay que desplazar empujándolos para poder alcanzar zonas más altas, objetos que se puedan recoger y lanzar, o piedras para armas arrojadizas o de asedio. En definitiva, las posibilidades son muchas y uno se puede hacer a la idea de qué objetos quiere que sean estáticos y cuales dinámicos.

En el videojuego de lucha implementado para la realización de este manual, casi todo es estático, ya que los personajes son humanos, y sería difícil mantenerlos de pie, y porque el escenario diseñado no tiene objetos con los que interactuar. Se han utilizado objetos dinámicos para crear las partículas de las bolas de energía lanzadas por los luchadores, las cuales necesitan disponer de un actuador de fuerza para ser disparadas en la dirección en la que está mirando el personaje. También se han utilizado objetos dinámicos para crear los objetos que deben provocar colisiones, como son los puños y los pies de los personajes.

Para hacer un objeto estático o dinámico, primero habrá que seleccionarlo, y después en la ventana "*3D Window*", seleccionar la sub-sección "*Logic*" y elegir el tipo de objeto en el siguiente menú desplegable:

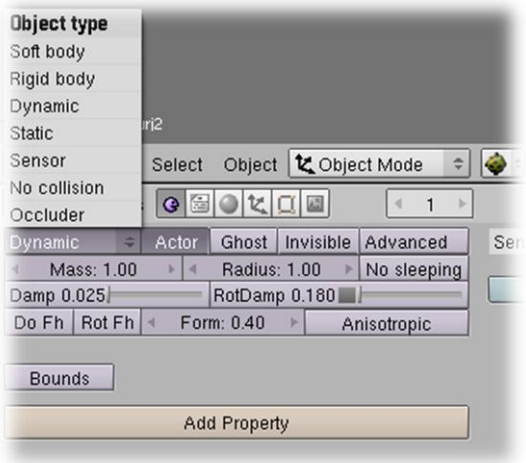


Ilustración 62. Menú tipo de objeto

Como se puede ver en el menú desplegable, aparte de objetos dinámicos y estáticos, se pueden crear otros tipos de objetos, como "*rigid body*", que se comporta como un objeto rígido real al que se puede golpear y no se deforma, o "*Soft body*", que se comporta como un objeto blando y deformable, que cambia su forma al posarse sobre una superficie o empujarlo. También existe el tipo "*Sensor*", que es un tipo que detecta especialmente bien las colisiones, y "*Occluder*", cuya función es que el juego no haga el renderizado de los objetos que estén detrás de los objetos de este tipo.

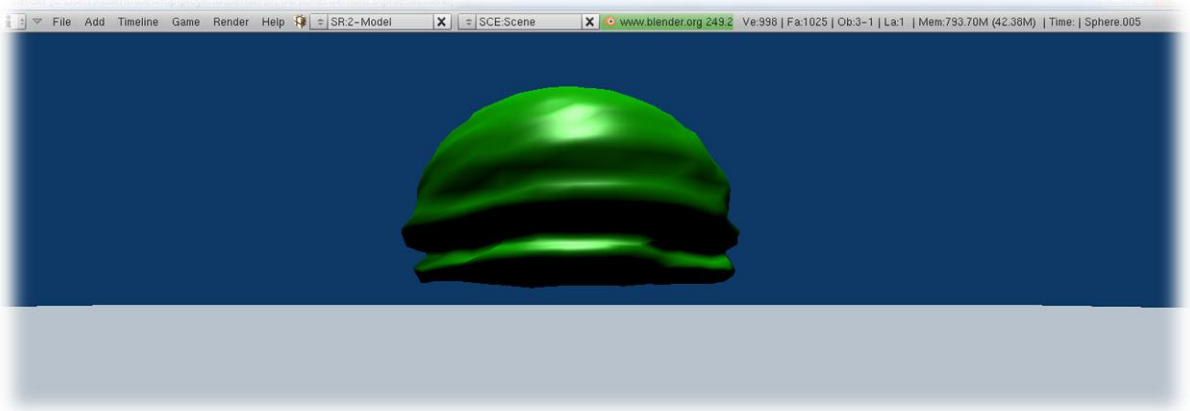


Ilustración 63. Ejemplo de esfera cayendo con tipo de objeto Soft Body

Los objetos de tipo "*soft body*" que estén emparentados con algún objeto, se comportarán como objetos rígidos.

4.5.Inserción de la lógica en *Blender* (*logic bricks*)

El modo de introducir lógica en *Blender* es muy peculiar, ya que se hace a través de un interfaz gráfico además de con código en *Python*. La idea básica que ofrece *Blender*, son una serie de bloques lógicos de tres tipos, los cuales se pueden combinar entre sí para crear la lógica de un videojuego. Estos bloques lógicos son sensores, controladores y actuadores y cada objeto tiene los suyos propios, dotando así de una mayor independencia a la posible inclusión de nuevos objetos en el juego.

Ilustración 64. Menú de bloques lógicos

El interfaz gráfico de *Blender* también permite crear propiedades o atributos para cada objeto. En la ilustración 64 se puede ver el botón *Add Property,* que permite crear una propiedad. Las propiedades tienen un nombre, un tipo y un valor, además de disponer de un botón al lado para depurar dicha variable. Encima de este botón se encuentran otras propiedades configurables para cada objeto, como pueden ser los distintos tipos de límites (*bounds*) de colisión, los tipos de objetos que ya se han explicado con anterioridad, y para cada tipo de objeto, parámetros como la visibilidad, la masa de los objetos dinámicos, o el radio de su superficie de colisión.

Todas estas propiedades y opciones configurables son muy influyentes a la hora de crear la lógica del juego ya que por ejemplo, hay sensores y actuadores basados en las propiedades creadas para el objeto. También existen sensores de colisión que dependen del tipo de superficie asignada como límite de colisión, del radio de dicha superficie, y de que el subtipo "*actor*" esté activado.

A continuación se comentarán más en detalle los controladores, actuadores y sensores, y como integrarlos con el código en *Python*. Después, se expondrán numerosos ejemplos implementados en el videojuego realizado para la creación de este manual, que facilitarán el entendimiento del mismo.

4.5.1. Sensores, controladores, actuadores y estados

El modo en que funcionan sensores, actuadores, controladores y estados es el siguiente: Un objeto puede estar en uno o varios estados a la vez, sensores y actuadores son compartidos por todos los estados de un objeto, pero los controladores sólo existen en uno de los estados a la vez.

Los elementos principales de la lógica son los controladores, a los que se deben conectar los sensores y actuadores que se quieran utilizar. Estas conexiones se realizan gráficamente, creando uniones mediante líneas con el ratón, como se puede ver en la ilustración 65. Mediante los controladores se puede cambiar de estado, para permitir que el objeto en cuestión pueda realizar más o menos acciones, o simplemente unas diferentes.

Ilustración 65. Ejemplo de unión de controladores, actuadores y sensores

Sensores

Los sensores de cada objeto se encargan de detectar determinados eventos que ocurren en su entorno. *Blender* dispone de una amplia variedad de sensores que se muestran en la ilustración 66, y que se definen a continuación en una tabla explicativa. Todos los sensores tienen unos parámetros en común, mostrados bajo el sensor "*Always*" que aparece en la ilustración 65.

El primer botón indica que se seguirán enviando pulsos positivos mientras, el sensor este activo, el segundo botón hace lo mismo, pero enviando pulsos negativos, el campo con la "f:" indica el numero de *"frames"* que se dejan pasar entre pulso y pulso, el botón "*Tap*" hace que el controlador sólo se active un instante, aunque el sensor siga activo, y el botón "*Inv*" invierte la salida del sensor.

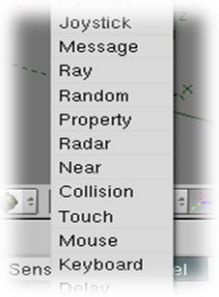


Ilustración 66. Sensores

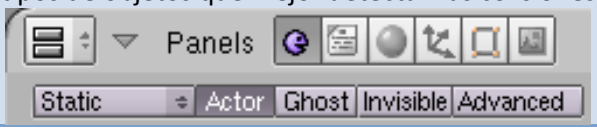
Nombre	Descripción
Actuator	Este sensor se dispara cuando el actuador cuyo nombre se ha insertado como parámetro se activa.
Joystick	Captura los diferentes movimientos de un *joystick.*
Message	Se activa al detectar un mensaje enviado por otro objeto, que tenga un sujeto determinado. Si al sensor no se le indica ningún sujeto, se activará al detectar cualquier mensaje.
Ray	Se activa cuando un objeto con la propiedad o material seleccionado como parámetro, se encuentra en un rango respecto del objeto propietario del sensor. El rango se indica también como parámetro del sensor.
Random	Se activa aleatoriamente. Se puede proporcionar una semilla como parámetro para generar la aleatoreidad.
Property	Se activa cuando la propiedad del objeto indicada como parámetro cumple: *1-* Es igual a un valor dado *2-* Es distinta a un valor dado *3-* Cambia de valor *4-* Está en un rango determinado
Radar	Se activa cuando un objeto con la propiedad seleccionada como parámetro, se encuentra en un rango respecto del objeto propietario del sensor, y dentro de un ángulo determinado también por un parámetro.
Near	Se activa cuando un objeto con la propiedad seleccionada como parámetro, está lo suficientemente cerca. La distancia a la que se considera un objeto cerca se indica como parámetro. El sensor sólo se activa una vez mientras el objeto permanezca cerca, para activarlo más veces, habrá que alejarlo una distancia determinada que también se indica como parámetro.
Collision	Se activa cuando un objeto con la propiedad seleccionada como parámetro choca o colisiona contra el objeto propietario del sensor. Hay que tener en cuenta que para que un objeto sea detectado debe tener el subtipo actor activado y que los tipos de objetos que mejor detectan las colisiones son los de tipo *Sensor.* 
Touch	Manda un pulso positivo al controlador si el objeto está en contacto con el material indicado como parámetro, y un pulso negativo en caso contrario.
Mouse	Captura los diferentes movimientos y *clicks* de un ratón.
Keyboard	Captura las pulsaciones de las distintas teclas de un teclado, combinaciones de las mismas, o el mantener pulsada una tecla. Además puede registrar las cadenas de texto que se introduzcan por teclado y almacenarlas en una propiedad indicada por parámetro.
Delay	Este sensor se activa una vez acabado un tiempo indicado por parámetro que empieza a contar cuando comienza el juego y dura también un tiempo determinado. También se puede activar la opción “REP” que hará que cada vez que se desactive el sensor vuelva a empezar la cuenta del tiempo que activa el sensor.
Always	Este sensor está activado continuamente.

Tabla 6. tabla de sensores

Controladores

Los controladores al igual que los sensores, tienen algunos parámetros en común, sean del tipo que sean. En la ilustración 67 se puede ver una estrella al lado del nombre del controlador, esta estrella significa que si está activada, ese controlador se ejecuta con mayor prioridad que los demás. El segundo parámetro indica el estado al que pertenece el controlador. Ese controlador sólo será utilizable si el estado del objeto coincide con el estado del controlador.

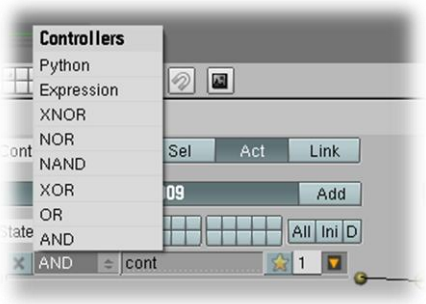


Ilustración 67. Controladores

Nombre	Descripción
Python	Este tipo de controlador, es un fichero escrito en *Python*, que puede recoger los valores de los sensores a los que está conectado, y activar los actuadores a los que también está conectado, en función del código que esté escrito en el script.
Expression	Evalúa, una expresión simple siempre que se active uno de los sensores conectados a él. En la expresión se pueden incluir tanto números como variables.
XNOR	Simula una puerta *XNOR* tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando el número de señales positivas de todos los sensores sea par.
NOR	Simula una puerta *NOR* tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando alguno de los sensores sea negativo.
NAND	Simula una puerta *XNOR* tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando todos los sensores sean negativos.
XOR	Simula una puerta XOR tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando el número de señales positivas de todos los sensores sea impar.
OR	Simula una puerta *OR* tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando alguno de los sensores sea positivo.
AND	Simula una puerta *AND* tomando los valores de los sensores como entradas para la expresión. Se activan los actuadores a los que esté conectado cuando todos los sensores sean positivos.

Tabla 7. Tabla de controladores

Actuadores

Los actuadores representan las acciones que pueden realizar los objetos, tales como movimientos, transformaciones, ejecución de animaciones o modificación de datos, y pueden ser ejecutados por los controladores a los que están conectados.

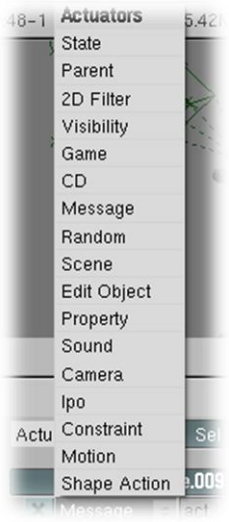


Ilustración 68. Actuadores

Nombre	Descripción
State	Añade o elimina uno o varios estados al grupo de estados activos.
Parent	Elimina o Asigna al objeto propietario del actuador, un padre indicado por parámetro.
2D Filter	Aplica distintos filtros artísticos a toda la escena.
Visibility	Hace al objeto visible e invisible.
Game	Inicia o acaba un juego.
CD	Realiza varias acciones sobre un CD de audio. Reproduce una o todas las pistas, modifica el volumen, pausa la reproducción o la detiene.
Message	Envía un mensaje a otro objeto. Opcionalmente se puede añadir un sujeto y un cuerpo al mensaje.
Random	Genera un número aleatorio a partir de una semilla, y lo almacena en la propiedad indicada por parámetro.
Scene	Realiza una acción sobre una escena, como por ejemplo, iniciar una escena, pausarla, reiniciarla, sobreponer una escena sobre otra, o cambiar la cámara activa para esa escena.
Edit Object	Permite añadir a la escena objetos ocultos en otras capas, eliminarlos, reemplazar una malla por otra, enfocar a un objeto y activar o desactivar las propiedades dinámicas del objeto. Si se elige la opción de añadir un objeto, habrá que indicar durante cuánto tiempo permanece ese objeto en escena, y si es dinámico, se le podrá aplicar una fuerza y una velocidad determinadas.
Property	Asigna un valor a una propiedad del objeto propietario de la misma.
Sound	Ejecuta un sonido.
Camera	Hace que la cámara enfoque a un objeto.
Ipo	Ejecuta una *IPO perteneciente* al objeto.

Constraint	Impone distintas restricciones al objeto, como pueden ser restricciones de movimiento, para que no salga de una zona especificada entre un máximo y un mínimo, que se mantenga a una distancia mínima de un punto, que se mantenga en una orientación determinada, o que se le aplique un campo de fuerza.
Motion	Puede cambiar la localización y orientación de un objeto en sus tres ejes. Si el objeto es dinámico, permite aplicar fuerzas de empuje sobre el mismo, además de velocidades lineales y angulares.
Shape Action	Puede ejecutar una de las acciones creadas previamente para un objeto de tipo esqueleto. Se debe indicar como parámetro el rango de *"frames"* que se quieren reproducir de la animación.

Tabla 8. Actuadores

Estados

Los estados en *Blender* se usan para permitir a un objeto limitar sus acciones de un modo rápido y fácil. En vez de usar código para decidir qué acciones puede realizar un personaje en cada momento, lo que se hace es crear en cada estado, los controladores que se puedan usar, y los actuadores que cambian de un estado a otro. En la ilustración 69 se pueden ver los 30 estados de un objeto en concreto, representados por pequeños cuadrados grises. El cuadrado sombreado representa el estado activo actualmente, y el circulo, representa al estado inicial del objeto.

Ilustración 69. Estados de los bloques lógicos

Por ejemplo, en el estado muerto, no se debería crear ningún controlador, ya que un personaje muerto no puede moverse ni hacer nada, en el estado salto, no se debería implementar el controlador que permite saltar al personaje a no ser que se quiera hacer un salto doble. Tampoco es lo mismo dar un puñetazo en estado de reposo, que cuando ya se está realizando algún otro ataque, ya que se podría dar lugar a distintas animaciones, como ocurre con los combos (combinaciones de golpes).

Si se utiliza un objeto para controlar la lógica global del juego, los estados también son muy útiles para diferenciar entre distintos momentos del juego, en los que cambie por ejemplo el clima, la gravedad, el tipo de música o el movimiento de la cámara. También pueden ser útiles al finalizar un nivel del juego, cambiar de pantalla, o ejecutar videos "*ingame*" que utilicen el motor gráfico del juego.

4.5.2. Scripts en Python

Para añadir un nuevo *script* escrito en *Python,* hay que abrir la ventana "*Text Editor", y* en el menú desplegable que contiene el campo "TX:" se selecciona "*ADD NEW"* como se puede observar en la ilustración 70. En el campo "TX:" se escribe el nombre del nuevo script. Los scripts no pertenecen a ningún objeto, y pueden ser usados por todos, es por eso que en el menú desplegable mencionado anteriormente aparecen todos los scripts creados para el juego.

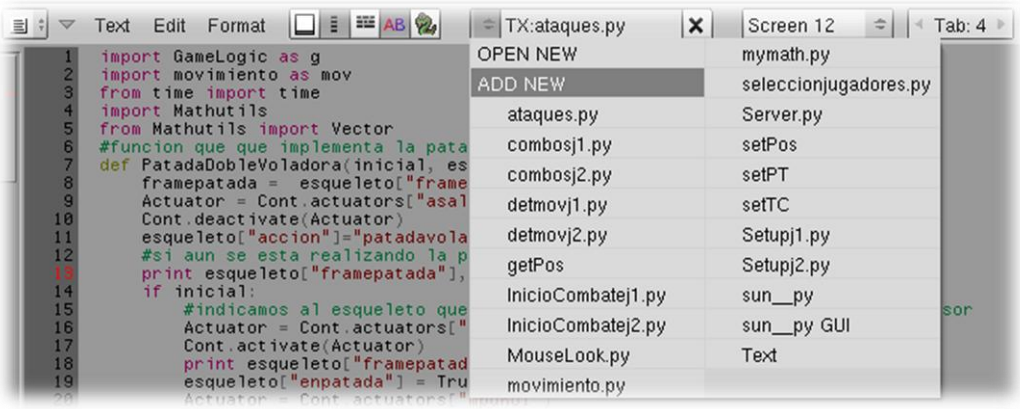


Ilustración 70. Editor de textos

Lo que un script hace en definitiva, es leer las señales recibidas de los sensores, y en función de ciertas condiciones que son las que aportan lógica al juego, ejecutará o no ejecutará los actuadores a los que esté conectado. Por esta razón a continuación se explica cómo obtener las señales de los sensores y como ejecutar un actuador en *Python*.

Para leer un sensor se utiliza el siguiente código:

```
##se obtene el controlador actual
Controlador = GameLogic.getCurrentController()
##se obtiene el objeto dueño del controlador
Objeto = Cont.owner
##se obtiene el sensor de nombre nombresensor
NombreSensor = Cont.sensors["nombresensor"]
##se obtiene la señal del sensor (True o False)
Senal=NombreSensor.positive
```

Ilustración 71. Código de obtención de sensores

Cuando un sensor se activa, puede enviar una señal positiva o negativa, y ambos valores pueden ser útiles, ya que el valor falso no significa que no se haya activado el sensor.

Para obtener, activar y desactivar un actuador se utiliza el siguiente código:

```
##se obtene el controlador actual
Controlador = GameLogic.getCurrentController()
##se obtiene el objeto dueño del controlador
Objeto = Cont.owner
##se obtiene el Actuador con nombre nombreactuador
Actuador = Cont.actuators["nombreactuador"]
##se activa el actuador
Cont.activate(Actuador)
##se desactiva el actuador
Cont.deactivate(Actuador)
```

Ilustración 72. Código de obtención y ejecución de actuadores

Si el tipo de controlador que utilizamos, no es un script en *Python*, los actuadores se activan y desactivan automáticamente, dependiendo de los parámetros asignados. Pero si activamos un actuador desde un script, con la función *activate(Actuador),* es necesario desactivarlo en el momento preciso con la función *deactivate*(Actuador).

Por ejemplo, si un script, activa la animación de saltar al pulsar la barra espaciadora, y no se desactiva posteriormente, el personaje seguirá saltando aunque no se esté presionando la barra espaciadora.

El resto de teoría relacionada con los scripts en *Python* es teoría de programación general, y por lo tanto no se explicará en este manual. En cambio, a continuación se expondrán numerosos ejemplos en los que se explicarán diferentes maneras de programar útiles en *Blender*, y útiles para el diseño de videojuegos. Los ejemplos mostrados a continuación, han sido sacados del juego implementado para la realización de este manual.

4.5.3. Ejemplos de aplicación de lógica

Ejemplo 1: inicialización de personajes seleccionados

Este script coloca a los personajes seleccionados por los jugadores en el ring de combate.

```
import GameLogic as g
Cont = g.getCurrentController()
SenInicial = Cont.sensors["inicioj1"]
if SenInicial.positive:
    GameLogic.pos1=[0,-3,0]
    GameLogic.pos2=[0,3,0]
    f = open("pj1.txt")
    contenido = f.read(1)
    f.close()

    #elegimos el personaje seleccionado por el jugador
    if contenido == "1":
        print "coloco j11"
        ColocarJugador11 = Cont.actuators["ColocarJugador11"]
        Cont.activate(ColocarJugador11)
    elif contenido == "2":
        print "coloco j12"
        ColocarJugador12 = Cont.actuators["ColocarJugador12"]
        Cont.activate(ColocarJugador12)
    g.Pj1 = -30
    owner = Cont.owner
    #owner.orientation=[0,0,3.141592]
    owner["ensalto"] = False
    owner["velocidadinicial"] = 0
    owner["accion"] = "parado"
    owner.localPosition = GameLogic.pos1
```

Ilustración 73. Código de inicialización de personajes

Este ejemplo proviene de un juego de lucha con un menú de selección de personajes. Mediante dicho menú se guarda en un fichero de texto el número de personaje que ha seleccionado cada jugador, y después con un controlador, se cambia a la escena en la que aparece el escenario con los luchadores. Los números de cada personaje se guardan en los ficheros "pj1.txt" y "pj2.txt", ya que *Blender* no mantiene las variables globales entre 2 escenas distintas.

Este script pertenece a la escena del combate, y lo que hace es leer el fichero "pj1.txt", y comprobar su contenido. Si su contenido es un 1 ejecuta el actuador que coloca al personaje número 1 en el lado izquierdo, y si es un 2 colocará al personaje número 2 en el lado izquierdo también.

Además de colocar a los personajes seleccionados en el menú inicial, inicializa algunos valores y crea variables globales como la posición inicial de los luchadores, la acción inicial, que es estar parado, y la velocidad inicial que es 0.

En la ilustración 74 se expone en un esquema cómo funciona el mecanismo para añadir 2 personajes a la escena de combate, seleccionando a "Fleur" para el primer jugador y a "Faccia" para el segundo.

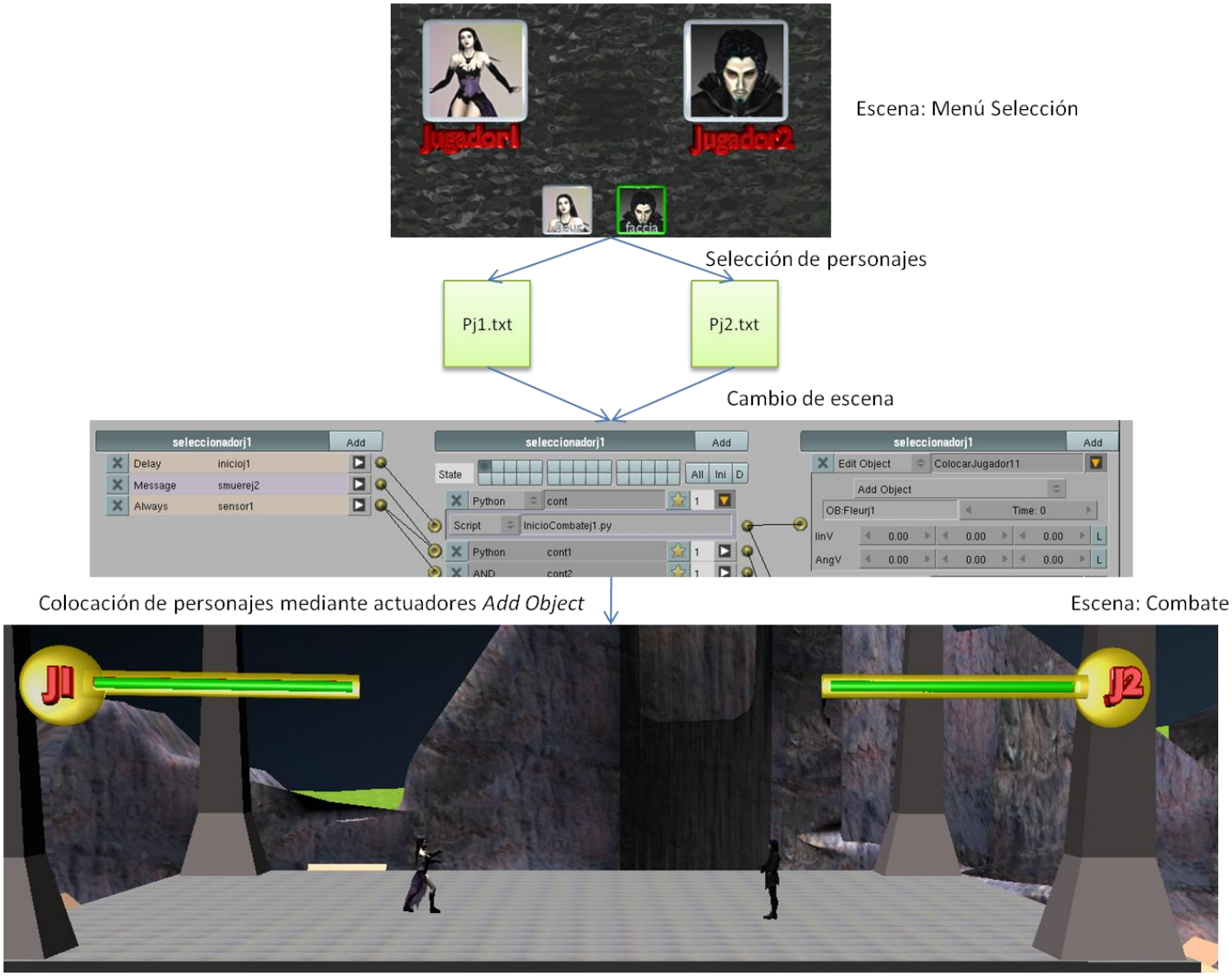


Ilustración 74. Diagrama de selección de dos personajes

En el diagrama se muestra como seleccionando a Fleur como primer jugador y a Faccia como segundo, se guardan sus números en los respectivos ficheros “Pj1.txt” y “Pj2.txt”, y como a partir del controlador “cont”, que es el script explicado en este apartado, se activa el Actuador “ColocarJugador11” que coloca a Fleur a la izquierda en el ring.

El objeto dueño de este controlador y actuadores, es un objeto vacío que está situado en el ring de combate, en la capa visible del mismo, y que mediante dicho actuador coloca al personaje seleccionado que existe en una capa oculta en su sitio correspondiente. Existe un objeto vacio para cada jugador en la capa visible, y en las capas ocultas hay una versión de cada personaje para cada jugador, es decir que hay 2 Fleur creadas y 2 Faccia, cada una con sus controladores, sensores y actuadores propios. De este modo cuesta un poco más añadir nuevos personajes, pero se facilita la implementación de los scripts.

Ejemplo 2: selección de ataque a realizar

Este script selecciona el ataque que el jugador pretende que realice el personaje luchador.

```
import GameLogic as g
from time import time
import Mathutils
import ataques as atk
import movimiento as mov
Cont = g.getCurrentController()
esqueleto = Cont.owner
SensorPatada = Cont.sensors["spatada"]
SensorPatadaVoladora = Cont.sensors["spatada2"]
SensorPatadaMultiple = Cont.sensors["spatada3"]
SensorEnPatada = Cont.sensors["senpatada"]
SensorPuno1 = Cont.sensors["spuno1"]
SensorPuno2 = Cont.sensors["spuno2"]
SensorBolaEnergia = Cont.sensors["sbola"]
SensorEnBolaEnergia = Cont.sensors["senbola"]

print "accion", esqueleto["accion"]
if ((SensorPatadaVoladora.positive and esqueleto["accion"]=="salto") or (esqueleto["accion"]=="patadavoladora")):
    atk.PatadaDobleVoladora(SensorPatadaVoladora.positive,esqueleto, Cont)
elif (SensorPuno1.positive or esqueleto["accion"] == "patadacombo"
     or esqueleto["accion"] == "puno1" or esqueleto["accion"] == "puno2"):
    atk.punetazo(esqueleto,Cont,SensorPuno1.positive)
elif ((SensorBolaEnergia.positive and (esqueleto["accion"]=="parado" or esqueleto["accion"]=="bolafuego"
    or esqueleto["accion"]=="doblesalto")) or (esqueleto["accion"]=="bolafuego")):
    atk.lanzar_bola(SensorBolaEnergia.positive, esqueleto, Cont)
elif ((SensorPuno2.positive and (esqueleto["accion"]=="parado" or esqueleto["accion"]=="avanza"
    or esqueleto["accion"] == "punofuerte")) or esqueleto["accion"] == "punofuerte"):
    atk.punetazofuerte(esqueleto,Cont,SensorPuno2.positive)
elif ((SensorPatada.positive and (esqueleto["accion"] == "parado" or esqueleto["accion"] == "agachado"))
    or ( esqueleto["accion"] == "patada1" or esqueleto["accion"] == "patadabaja")):
    atk.patada(SensorPatada.positive, esqueleto, Cont)

#comprobacion de tiempo de combos
g.horaUltimoGolpe = time()
g.horaultimomov = time()
```

Ilustración 75.Ejemplo de selección de ataques

Al comienzo del código se importa una librería que no pertenece a las librerías de *Blender*. Esta librería es un script implementado por el autor del proyecto y se llama “ataques”. Contiene funciones que ejecutan los distintos ataques que pueden realizar los personajes.

Para decidir que ataque o movimiento se va a realizar, primero se obtienen los sensores que están conectados a este controlador, y que representa la interconexión con el interfaz de usuario. Conociendo la tecla pulsada por el jugador y el estado en el que se encuentra en el momento de pulsarla, se sabe a qué función se debe llamar. Por ejemplo si la acción que está realizando un personaje es saltar, y se pulsa la tecla de puñetazo, la orden de puñetazo se deberá ignorar, y el sato deberá acabar normalmente. En cambio, si se está saltando y se pulsa la tecla de patada voladora, en medio del salto se tendrá que ejecutar la animación de dar una patada estando en el aire.

Es necesario aclarar que el que se ejecute una misma función no siempre llevará a ejecutar el mismo ataque. Por ejemplo si se pulsa el puñetazo, y se vuelve a pulsar el puñetazo en el momento preciso, antes de que el primero acabe, se ejecutará una animación que representa un combo formado por un puñetazo derecho y uno izquierdo. Esto se consigue mediante los parámetros pasados a las funciones, que son principalmente, El objeto controlador, el objeto dueño del controlador, que es en este caso el esqueleto, y la señal del sensor en cuestión que active esa función. En los siguientes ejemplos se explica cómo usar estos parámetros para modificar el comportamiento de las funciones.

Ejemplo3: uso de "frames" para control de la lógica

Este script, es la función que ejecuta el lanzamiento de la bola de energía, que al ser una animación compleja y que involucra a varios objetos, necesita una lógica un poco más compleja.

```
#funcion que que implementa la bola de energia
def lanzar_bola(inicial, esqueleto, Cont):
    ##guardamos en una variable el frame por el que va la animacioon
    framebola =  esqueleto["fbola"]
    ##dependiendo del frame en el que este la animacion seleccionamos la accion a realizar
    if esqueleto["accion"] == "bolafuego":
        if framebola == 5:
            Actuator = Cont.actuators["mbs"]
            Cont.activate(Actuator)
        elif framebola == 30:##se inicia la bola grande
            Actuator = Cont.actuators["bolagrande"]
            Cont.activate(Actuator)
        elif framebola == 90:##se empiezan a disparar bolas
            Actuator = Cont.actuators["lanzamientobola"]
            Cont.activate(Actuator)
            Actuator = Cont.actuators["bolagrande"]
            Cont.deactivate(Actuator)
        elif framebola == 189:
            Actuator = Cont.actuators["lanzamientobola"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["afinbola"]
            Cont.activate(Actuator)
        elif framebola == 199:##se finaliza el lanzamiento de la bola
            esqueleto["enbola"] = False
            esqueleto["accion"]="parado"
            Actuator = Cont.actuators["fmbs"]
            Cont.activate(Actuator)
            Actuator = Cont.actuators["lanzarbola"]
            Cont.deactivate(Actuator)
    else:##se inicia el lanzamiento de la bola de energia
        esqueleto["accion"] = "bolafuego"
        Actuator = Cont.actuators["lanzarbola"]
        Cont.activate(Actuator)
```

Ilustración 76. Código de uso de *"frames"* para el control de la lógica

La primera vez que se ejecute la función, el hilo de ejecución entrará en el *else*, y la acción del esqueleto pasará a ser "bolafuego". Además se activará la animación que realiza el esqueleto para lanzar la bola de energía.

El resto de veces que se ejecute la función, el hilo de ejecución entrará en el primer *if*, y dependiendo del *"frame"* por el que vaya la animación, se irá metiendo en los distintos *ifs interiores, que van ejecutando distintos actuadores.*

El primero de estos actuadores, hace aparecer una bola de energía pequeña sobre la cabeza del personaje.

Ilustración 77. Bola pequeña

El segundo envía un mensaje a la bola creada en primer lugar para que esta crezca, rote y reproduzca un sonido.

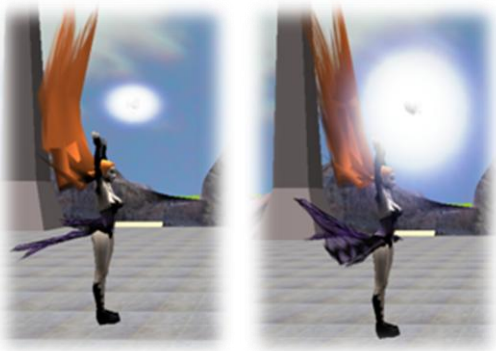


Ilustración 78. Crecimiento bola

El actuador "lanzamientobola", crea continuamente bolas medianas de energía, y las lanza hacia adelante, creando el efecto de un rayo de energía.

Ilustración 79. Ejemplo lanzamiento bola de energía

El actuador "afinbola", envía un mensaje al objeto vacío que lanza las bolas, para que deje de lanzarlas, y para que la bola grande que aparece pegada a las manos desaparezca también.

Puesto que la creación y transformación de las distintas bolas, depende del *"frame"* de la animación del esqueleto, guardando este *"frame"* en una variable, tal y como se hace en el script de este apartado, se pueden sincronizar animación y actuadores.

Para almacenar el *"frame"* actual de una animación en una variable, basta con poner como parámetro del actuador que ejecuta dicha animación, el nombre de la propiedad en la que se quiere almacenar el número de *"frame"*. La propiedad usada a modo de ejemplo en la ilustración 80 es "*fbola*".

Ilustración 80. Propiedad para almacenar número de *"frame"*

Ejemplo4: implementación de combos

Esta función pertenece al script "ataques.py" e implementa el combo que se produce al presionar 1, 2 o 3 veces la tecla de puñetazo.

```
#funcion que que implementa los puñetazos
def punetazo(esqueleto, Cont,sensor):
    if esqueleto["accion"] == "parado":#primer puñetazo del combo
        esqueleto["accion"] = "puno1"
        Actuator = Cont.actuators["apuno1"]
        Cont.activate(Actuator)
        Actuator = Cont.actuators["mpuño1"]
        Cont.activate(Actuator)
    elif esqueleto["accion"] == "puno1" and sensor:#segundo puño del combo
        if(esqueleto["framepuño"] >=5):
            esqueleto["accion"] = "puno2"
            Actuator = Cont.actuators["apuno1"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["apuno2"]
            Cont.activate(Actuator)
            Actuator = Cont.actuators["mpuño1"]
            Cont.activate(Actuator)
    elif esqueleto["accion"] == "puno2" and sensor:#tercer golpe del combo
        if(esqueleto["framepuño"] >= 8):
            esqueleto["accion"] = "patadacombo"
            Actuator = Cont.actuators["apuno2"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["apatadacombo"]
            Cont.activate(Actuator)
            Actuator = Cont.actuators["mpuño1"]
            Cont.activate(Actuator)
    elif esqueleto["accion"] == "patadacombo":#final del tercer golpe
        if(esqueleto["framepatada"] == 19):
            Actuator = Cont.actuators["apatadacombo"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["mfinpuño"]
            Cont.activate(Actuator)
            esqueleto["accion"] = "parado"
    else:
        if esqueleto["accion"] == "puno1" and esqueleto["framepuño"] == 15:#final del primer golpe
            Actuator = Cont.actuators["apuno1"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["mfinpuño"]
            Cont.activate(Actuator)
            esqueleto["accion"] = "parado"
        elif esqueleto["accion"] == "puno2" and esqueleto["framepuño"] == 15:#final del segundo golpe
            Actuator = Cont.actuators["apuno2"]
            Cont.deactivate(Actuator)
            Actuator = Cont.actuators["mfinpuño"]
            Cont.activate(Actuator)
            esqueleto["accion"] = "parado"
```

Ilustración 81. Combo de puñetazos

En el ejemplo 2 de este mismo apartado, se llama a esta función siempre que se presione la tecla de puñetazo, o el estado del personaje sea, "puno1", "puno2" o "patadacombo".

Si se presiona una vez el puñetazo, y el esqueleto está parado, da un puñetazo normal y se entra en estado "puno1" mientras la animación de este dure. Si se presiona el puñetazo, mientras se está en el estado "*puno1*" y la animación del primer puñetazo, ha reproducido más de 5 *"frames"*, se pasa al estado "*puno2*" y se reproduce la animación del puñetazo con el brazo izquierdo. Hay que esperar hasta el *"frame"* 5 para que el primer puñetazo no se vea solapado por el segundo, y la consecución de las dos animaciones no se vea con saltos.

Por último si se está en el estado "*puno2*", y se presiona la tecla de puñetazo de nuevo, y después del *"frame"* 8 de la animación del segundo puñetazo, el luchador girará sobre sí mismo para golpear con el reverso de la mano al contrincante. Al final, esté en el estado que esté el luchador, se deberá desactivar el actuador que esté activado cuando acabe la animación.

4.6.Detección de colisiones

La detección de colisiones es imprescindible en cualquier motor de juego hoy en día, y *Blender*, gracias a su motor de físicas, posee un avanzado sistema de detección de colisiones que se puede utilizar de diversas maneras.

El primer uso que se puede dar a la detección de colisiones, es el de que un objeto dinámico se apoye sobre una superficie estática sin atravesarla. Para probar este tipo de colisiones bastaría con crear un plano y una esfera, asignar el tipo "*Dynamic*" o "*Rigid body*" a la esfera, e iniciar el motor de juego pulsando la tecla "p". Al hacerlo se vería como la esfera cae sobre el plano y se mantiene sobre él. Si el plano estuviese inclinado, la esfera rodaría sobre el mismo hasta caer por uno de sus lados. Otra posible aplicación de este tipo de colisiones, es que el personaje no pueda atravesar paredes y en general cualquier objeto que en la realidad sería solido.

Otro uso que se puede dar a las colisiones, es el de activar eventos cuando se detecta una colisión, ya sea por recibir un golpe, llegar a un lugar, o tocar un objeto. Por ejemplo, si se quiere activar un elevador apretando un botón o abrir una puerta al tocar el pomo, cuando el botón o el pomo detecten una colisión, cada uno enviará un mensaje a la puerta o el elevador, y cuando estos los reciban, activarán sus respectivas animaciones de apertura de puerta o activación del elevador.

Si los dos objetos que colisionan son dinámicos, no será necesario utilizar bloques lógicos, para provocar la reacción de uno de los objetos al ser golpeado por el otro. Por ejemplo, si tenemos dos cubos dinámicos, y podemos mover uno de ellos con el teclado, cuando le acerquemos al otro cubo, y entren en contacto, el primero empujará al segundo sin necesidad de haber introducido lógica de colisiones a ninguno de los dos.

Un objeto no detecta las colisiones cuando otro impacta en su superficie, sino que las detecta cuando un objeto colisiona con sus límites de colisión o "*bounds*". Estos límites de colisión pueden ser formas básicas, como esferas y cajas, pero también pueden estar construidos mediante mallas más complejas. En la ilustración 82 se muestra el menú en el que se elije el tipo de límite para un objeto y que se encuentra en el icono "*Logic*" de la ventana "*Buttons window*".

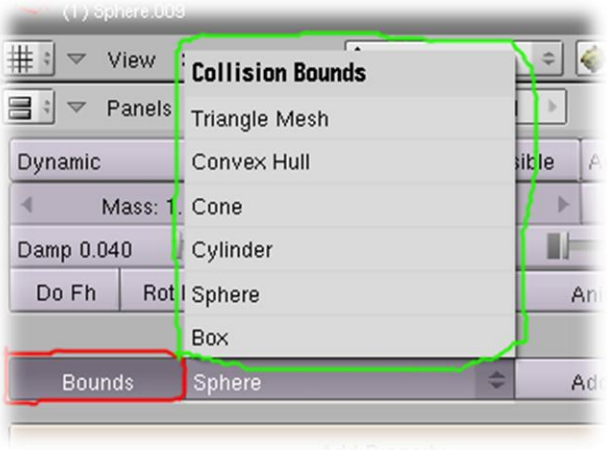


Ilustración 82. "*Bounds*"

Si el tipo de límite seleccionado es "*Convex Hull*" o "*Triangle Mesh*", estos límites no se podrán ver físicamente en la ventana "*3D Window*", pero si el tipo elegido es cualquiera de las otras formas, estas formas se podrán ver representadas en la pantalla seleccionando el tipo de visión "*Bounding Box*".

Hay ocasiones en las que no se detectan bien las colisiones con objetos estáticos. Para estas ocasiones lo mejor es utilizar objetos de tipo sensor para detectar dichas colisiones. Esta situación puede darse por ejemplo cuando la malla que mueve un esqueleto es un objeto estático, y queremos que detecte las colisiones que recibe. Para solucionar esto con objetos de tipo sensor, es posible crear un objeto invisible de tipo sensor para cada zona que queremos que se detecte la colisión.

Por ejemplo, en el juego implementado para realizar este manual, no es la malla del personaje la que detecta las colisiones, y esto no sólo es por causa de que no las detecte bien, si no porque interesa tener zonas de colisión diferenciadas, para detectar cuando se da un puñetazo, cuando se recibe un golpe, y en que parte del cuerpo se recibe dicho golpe.

En la ilustración 83 se pueden ver todos los objetos de tipo sensor que rodean a "*Fleur*". Normalmente el material de estos objetos, es totalmente transparente ya que no se deben ver durante la ejecución del juego, pero en esta ocasión se les ha asignado un material semitransparente para que se puedan ver en la imagen.

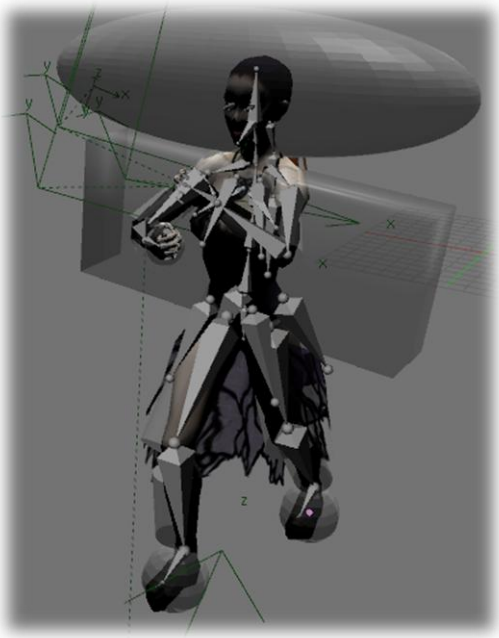


Ilustración 83. Áreas de colisión

La ilustración 84 representa lo mismo que la imagen anterior, sólo que en esta lo que estamos viendo son los límites de colisión de todos los objetos que hay en la pantalla. Se puede observar que hay una caja grande que cubre todo el cuerpo de la luchadora, esto es debido a que todos los objetos tienen asignados límites de colisión por defecto. Si se trata de un objeto estático, y no le añadimos ningún tipo de lógica que trate las colisiones, será como si sus límites de colisión no existiesen.

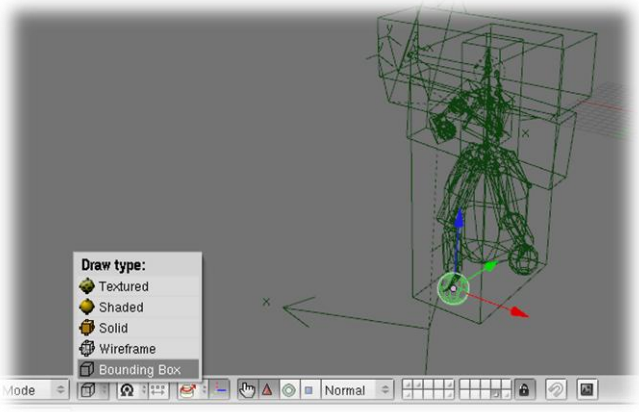


Ilustración 84. Vista "*Bounding box*"

En el juego de lucha implementado, las esferas de los puños y los pies se utilizan para crear los efectos de luz típicos de este tipo de videojuegos al golpear al enemigo, de modo que el efecto de luz aparezca siempre donde golpee el puño o el pie.

Ilustración 85. Efecto de luz en golpe

El resto de zonas se utilizan para detectar los golpes recibidos, y ejecutar la animación correspondiente. Por ejemplo, si "*Fleur*" recibe un golpe en las piernas, caerá al suelo de lado, si recibe un puñetazo fuerte en el abdomen, se inclinará hacia adelante quejándose, y si recibe un golpe en la cabeza, caerá violentamente hacia atrás.

Ilustración 86. Golpe bajo

Ilustración 87. Golpe en la cabeza

Ilustración 88. Golpe fuerte en estomago

4.7.Inserción dinámica de objetos en escenas

En *Blender* se pueden añadir objetos de forma dinámica a la escena de un juego. Para añadir un objeto a una escena durante la ejecución del juego, son necesarios varios requisitos:

- El primero es que dicho objeto exista en una de las capas ocultas de la escena.
- El segundo es que en las capas visibles de la escena, exista un objeto que sea el encargado de crear el nuevo objeto.

Para crear un objeto en la capa visible con la ayuda de otro, sólo será necesario ejecutar un actuador de tipo "*Edit Object*". Los parámetros de ese actuador deberán ser: "*Add Object*" en el menú desplegable, el nombre del objeto que se quiere añadir, y el tiempo que queremos que ese objeto permanezca en escena. Si el tiempo seleccionado es 0, el objeto aparecerá permanentemente. Opcionalmente, se puede asignar una velocidad lineal o angular inicial, al objeto que se añade a la escena.

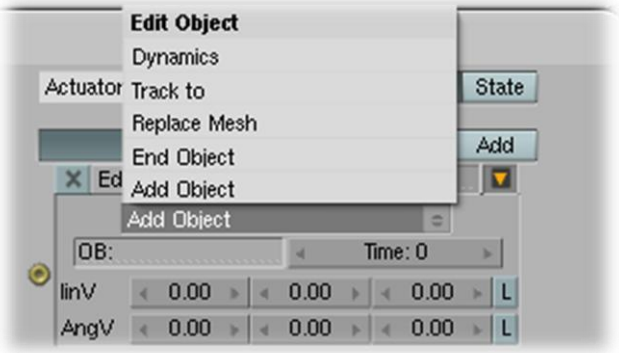


Ilustración 89. Inserción dinámica de objetos

Esta técnica es muy útil para realizar un gran número de acciones, creación de efectos especiales, replicación de personajes, obtención de nuevos objetos que se pueden encontrar por el escenario o aparición de menús sobre la pantalla.

A modo de ejemplo, en el juego realizado para la creación del manual, se ha utilizado esta técnica para crear las bolas de energía del ataque especial de la luchadora, se ha utilizado también para crear los efectos de luz al golpear al enemigo, y para añadir al comienzo del combate, los luchadores seleccionados en el menú. Otro ejemplo de aplicación podría ser el de crear una salpicadura de agua en una superficie acuática cuando un objeto cae dentro.

En apartados posteriores se explicará cómo utilizar esta técnica para crear efectos especiales que a porten realismo al videojuego.

4.8.Implementación de un menú dinámico de selección de personajes

En este apartado se explicará cómo crear el menú diseñado en el videojuego implementado para escribir este manual, que será extrapolable para diseñar una gran variedad de menús.

Ilustración 90. Menú de selección de personajes

Para crear el menú mostrado en la imagen superior, se han seguido 4 pasos:

1. *Colocación de la cámara y el fondo:*

   El primer paso que hay que llevar a cabo, es colocar la cámara paralela al plano XY. Esto se hace para saber que parte de la escena aparecerá en pantalla cuando se ejecute el juego, y poder colocar el resto de objetos adecuadamente.

   De modo opcional, se puede añadir un fondo a la pantalla, para darle un toque más profesional. El método es muy sencillo, se coloca un plano paralelo al plano XY, y se le aplica la textura o material que se quiera, cuanto más elaborado sea el material mejor aspecto visual dará. También se puede crear el fondo con objetos 3D que se muevan para animarlo un poco, siempre teniendo en cuenta los objetos que deberán incluirse obligatoriamente para dar funcionalidad al menú.

   Los siguientes pasos constan en añadir los diferentes tipos de objetos funcionales del menú, y explicar la lógica aplicada a los mismos.

2. *Creación de las letras:*

   Para que el jugador sepa cuál es su personaje, es necesario escribir con letras su nombre debajo del marco que contendrá la imagen de su luchador seleccionado. Para poder crear objetos 3D con forma de texto hay que acceder a la ventana "*Text Editor*", escribir el texto en el editor, y en el menú "*Edit*", seleccionar "Text to 3D Object".

   También se puede pulsar la barra espaciadora en la ventana "3D Window" y seleccionar "Text". Estos dos métodos crearán un objeto 3D de texto que posteriormente podrá ser modificado en la ventana "*Buttons Window*", en el icono "*Editing*". Aquí aparecerán varias herramientas que permiten editar el texto 3d, herramientas como cambiar la fuente elegida, dar

curvatura al texto, modificar su achura y profundidad, o biselar los bordes de las letras. Hay que tener en cuenta que este objeto no es una malla, y que por lo tanto no se puede modificar como tal. Para convertir el texto en malla, hay que seleccionarlo, pulsar "Alt+C" y seleccionar "*Mesh*".

3. *Creación de los marcos y las imágenes:*

Crear los marcos es sencillo, se crean tantos cubos como marcos vayamos a necesitar. En este caso 2 grandes, uno para cada jugador, y 2 pequeños uno para cada luchador existente en el juego.

Se modifican los marcos hasta que queden a gusto del diseñador y se crea un tercer marco pequeño idéntico a los dos anteriores, pero de un color distinto. Este marco se colocará fuera del campo de visión de la cámara y servirá para saber sobre la imagen de que luchador de la lista de luchadores, está el foco en cada momento.

Además se deberán crear 2 planos pequeños para colocar la lista de luchadores disponibles, y fuera del campo de visión de la cámara otros 2 planos más grandes. Estos planos representarán cada luchador disponible en el juego, y se irán colocando en los marcos del jugador 1 y jugador 2, a medida que el foco vaya desplazándose por la lista de luchadores.

Ilustración 91. Marcos del menú

Para reemplazar un marco gris por otro verde y viceversa, o para remplazar los planos con las imágenes de los luchadores en los marcos grandes a medida que se avanza por la lista, se utilizará un actuador de tipo "*Edit Object*". En este actuador se seleccionará la función "*Replace Mesh*", y en el campo "*ME:*" se pondrá el nombre de la malla por la que se quiere sustituir.

Cuando se sustituye una maya por otra, se sustituye la malla pero no el objeto, con lo que este sigue conservando todas sus propiedades, tanto lógicas como físicas. Gracias a esta propiedad podremos añadir una lógica sencilla al objeto marco, en la que cada vez que el foco esté sobre la imagen de un luchador, el marco de dicha imagen se vuelva verde, y en caso contrario que se vuelva gris.

4. *Funcionamiento de la lógica del menú*

En este ejemplo los jugadores elijen personaje uno a uno, primero el jugador 1 y luego el jugador 2. Con las flechas del teclado navegan por la lista de luchadores, y cuando están sobre el personaje con el que desean jugar, pulsan la tecla "enter" para seleccionarlo.

Cuando el primer jugador elige luchador, pasa al turno del jugador 2, y cuando este elige jugador, se cambia de escena al ring de combate, mediante un Actuador de tipo "Scene".

Sobre los marcos grandes, se colocan unos objetos de tipo "Empty", que recibirán un mensaje desde la lista de luchadores, con el identificador del luchador que está seleccionado en cada momento. Cada vez que cambiemos la selección en la lista, se enviará un mensaje al objeto "*empty*" del marco grande, y este colocará la malla con la imagen más grande del luchador cuyo identificador se ha recibido por mensaje. Es decir que en el marco grande del jugador, se mostrara en cada momento el luchador en el que esté situado.

Cuando se pulse la tecla enter, se guardará en una propiedad del objeto "*Empty*" el luchador seleccionado, y se cambiará al estado 2 de los bloques lógicos.

En este estado los mensajes ya no se enviarán al objeto "*Empty*" del jugador 1, si no que se enviarán al del jugador 2. A continuación se aplicará la misma lógica que con el primer jugador, y al pulsar la tecla "enter" se cambiará de escena y comenzará el juego.

4.9.Implementación de un controlador configurable y reutilizable

Implementar un controlador en *Blender* para un personaje, es muy sencillo. Sólo hay que seleccionar al personaje, y crear sensores de tipo teclado, ratón o joystick para cada tecla o movimiento que se quiera realizar. Este método, pese a ser muy sencillo, implica algunos problemas de eficiencia. Si se piensa por ejemplo en un juego de lucha, en el que hay 10 posibles personajes a elegir, y en el que además se implementa un menú que permite configurar los controles de cada jugador, habría que crear una lógica que modificase los sensores de cada luchador, dependiendo de qué jugador los maneje. Es decir que si se tienen 10 luchadores y dos jugadores, el programa debería ser capaz de modificar 10 objetos.

Para simplificar este proceso de configuración, se puede crear un objeto que haga de controlador para cada jugador, a los que se trasladarán los sensores que de otro modo pertenecerían a cada uno de los luchadores. Los sensores situados en los luchadores serán de tipo mensaje, y habrá tantos como botones tenga el controlador. Cada uno detectará la llegada de un mensaje con un sujeto distinto. De este modo, la lógica de control de todos los luchadores es exactamente idéntica y no será necesario cambiarla al cambiar su configuración.

En la ilustración 92 se muestra un diagrama que resume el proceso de configuración de los controladores. Desde el recuadro azul, que representaría el menú de configuración, se realizan las modificaciones deseadas sobre los controles por defecto del juego. Los cuadrados verdes, que representan a los controladores, quedarían con sus sensores configurados del modo descrito en el diagrama. Los cuatro cuadros rojos, representan los luchadores disponibles en el juego, la configuración descrita encima de ellos, es común para todos, y representa los sensores pertenecientes a cada luchador.

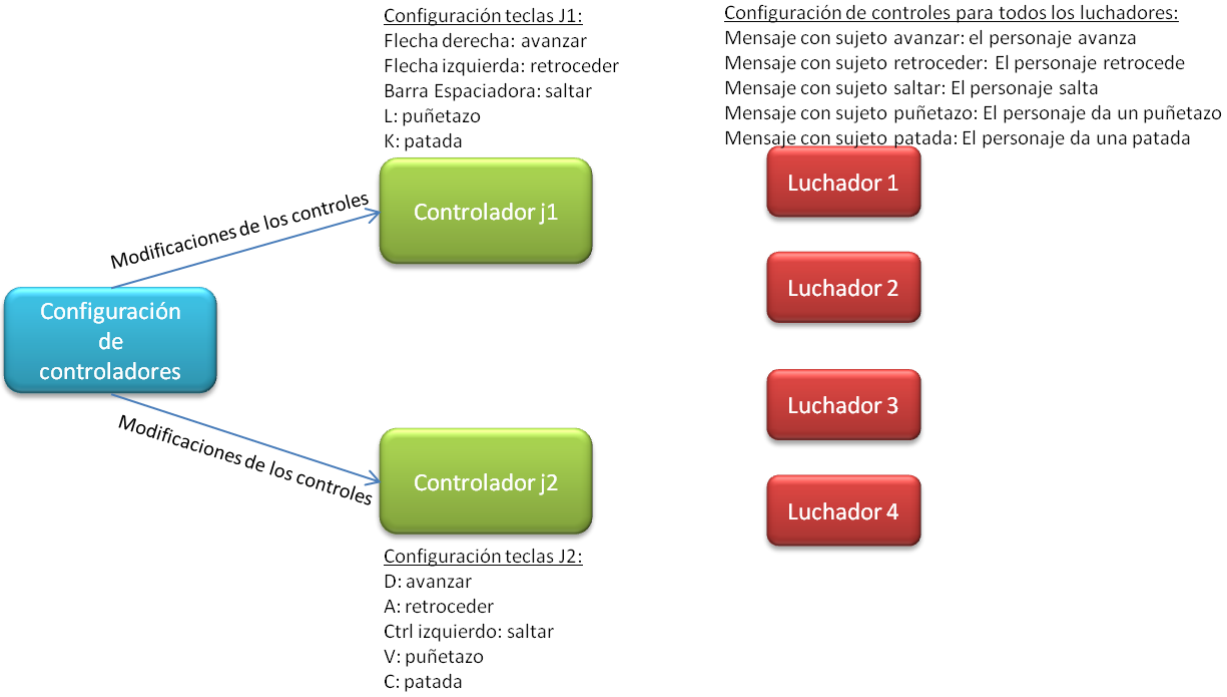


Ilustración 92. Ejemplo de configuración de los controladores.

En la ilustración 93, se muestra un diagrama que representa un ejemplo de funcionamiento le los controladores.

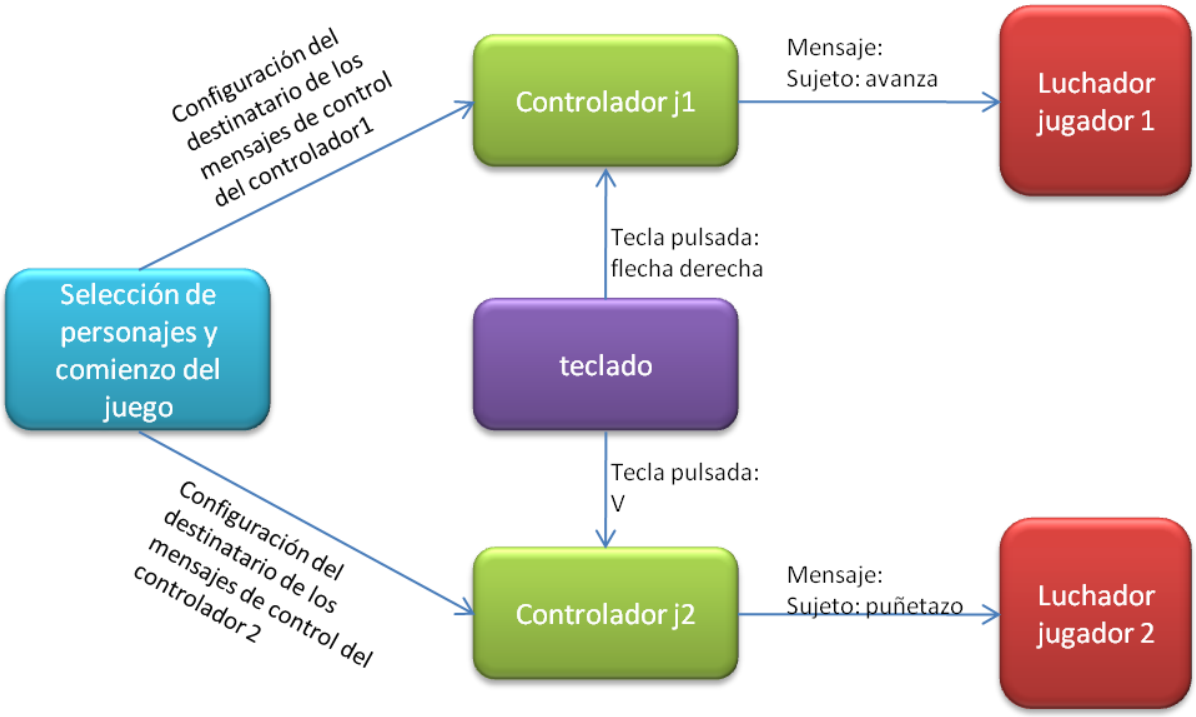


Ilustración 93. Diagrama de funcionamiento de los controladores.

El cuadro azul representa el menú de selección de personajes, una vez los jugadores hayan seleccionado a sus luchadores, se cambiará al escenario de lucha, y se indicará a los controladores, cuales son los personajes a los que tienen que mandar los mensajes de control.

Los cuadros verdes representan a los controladores, que recibirán los eventos capturados del teclado representado con un cuadro morado, y enviarán un mensaje con la acción a realizar a sus luchadores correspondientes.

En este ejemplo, el controlador del jugador 2 ha enviado el mensaje “puñetazo” al luchador para que este ejecute un ataque, porque se ha pulsado la tecla V, pero si el controlador se hubiese configurado con otra tecla distinta, el mensaje enviado al luchador habría seguido siendo “puñetazo”.

Viendo cómo funciona la configuración y la ejecución de los controladores, sólo será necesario modificar los parámetros de los sensores de los 2 objetos controladores mediante un script en *Python*, en vez de tener que modificar los sensores de todos los luchadores. Los actuadores no se modifican, y por tanto aunque cambien las teclas de control, los mensajes enviados a los luchadores, serán los mismos.

4.10. Implementación de una barra de vida para un personaje

En este apartado se va a explicar cómo se ha creado la barra de vida del juego de lucha, incluyendo diseño, animación y lógica.

4.10.1. Diseño

Una barra de vida o de salud, podría diseñarse de muchas maneras, pero aquí se explica la más simple de y típica de todas, que podría ser aplicable a cualquier tipo de juego.

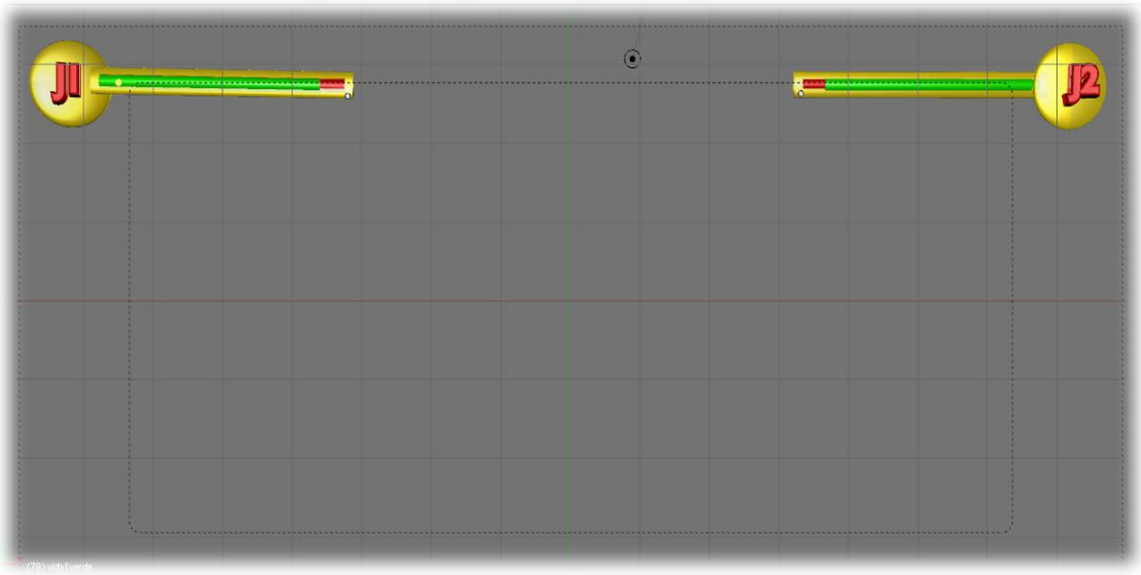


Ilustración 94. Ejemplo de barras de vida

Para empezar hay que crear una nueva escena, y añadir una cámara perpendicular al eje Z, y enfocando al plano XY, al igual que se hizo para crear el menú de selección de personajes. Después hay que ponerse en la vista de la cámara, pulsando el número 0 en el teclado numérico. La barra más básica se puede hacer con 2 planos alargados en forma de rectángulo. Primero se crea un plano rectangular rojo dentro de los límites de la cámara. Los límites de visión de la cámara es la línea punteada exterior que se puede ver en la ilustración 94. Encima del plano rojo hay que hacer un plano exactamente igual, pero de color verde, esto representará la vida que le queda al personaje.

Una vez se han hecho los planos rojo y verde, se le pueden añadir objetos decorativos, como los círculos que aparecen en la imagen, con las letras que indican a los jugadores propietarios de cada barra de vida, o sus imágenes representativas. También se puede añadir una barra que recubra a las otras dos por los laterales, o usar cilindros en vez de planos, que causan unos efectos de reflexión de la luz más llamativos, como pasa en el ejemplo superior. Igual que se han utilizado 2 rectángulos para crear las barras de vida, se podría utilizar cualquier otro tipo de geometría para indicar la disminución de vida, o incluso cambios graduales de color o animaciones de la cara de los personajes. Como siempre las decisiones de diseño 3D quedan a gusto del diseñador.

4.10.2. Animación

Para hacer que la barra de vida verde disminuya hay que seguir los siguientes pasos:

1. es necesario crear una "IPO" que escale la barra de vida verde hacia el extremo de la pantalla. Para hacer esto, lo primero que hay que cambiar en la ventana "*Buttons Window*" es el pivote a partir del que se va a hacer esta transformación.

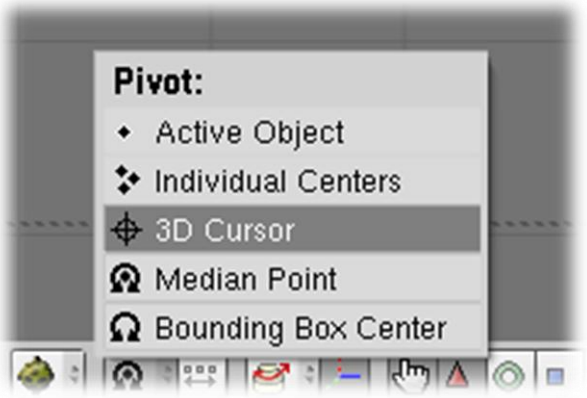


Ilustración 95. Pivote "3D Cursor"

2. Seleccionar como pivote el cursor 3D.
3. Colocarse de nuevo en la vista de la cámara y colocar el cursor en el extremo de la barra verde hacia el que va a disminuir esta.
4. Almacenar en el *"frame"* 100 de la "*IPO*", el rectángulo tal cual.
5. Almacenar en el *"frame"* 0 de la "*IPO*", el rectángulo escalado hasta el punto de que prácticamente no se vea.
6. Añadir una propiedad a la barra que se llame por ejemplo vida, y asignarle el valor inicial 100.
7. Añadir un actuador de tipo IPO con el parámetro "*property*" activado, y en el parámetro "*Prop*", escribir el nombre de la variable vida creada previamente. De este modo, el *"frame"* en cada momento de la IPO del objeto, será el que marque la propiedad vida.

Ilustración 96. IPO barra de salud

4.10.3. Lógica

Para que la vida del personaje aumente o disminuya, habrá que añadir sensores de tipo mensaje, para cada tipo de evento que se quiera que modifique la vida del personaje. Por ejemplo si recibe el mensaje puñetazo, que reste 2 a la propiedad vida, y si recibe el mensaje corazón, que aumente en 10 la propiedad vida. Entonces cuando un personaje reciba un puñetazo, su propiedad vida se

reducirá a 98, y en consecuencia la barra de vida verde se reducirá una pequeña cantidad, un 2% para ser más concretos.

Por último, hay que recordar que todos estos objetos están situados en una escena distinta a la escena del combate, y hay que colocarlos en dicha escena de modo que siempre queden en la misma posición respecto a la cámara. También hay que colocar las barras de vida intentando que queden siempre visibles por encima de los demás objetos.

Todo esto se puede hacer con un simple actuador de tipo "*Scene*", configurado con el parámetro "*Overlay Scene*", sobreponer escena, tal y como está configurado en la imagen inferior.

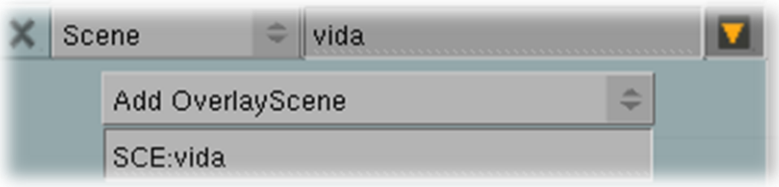


Ilustración 97. Sobre posición de escenas

Este actuador se tendrá que colocar en un objeto perteneciente a la escena en la que se quiere que aparezca, y conectado con un controlador y un sensor de tipo "*Delay*", que sólo se ejecute una vez.

Con este actuador se consigue sobreponer la imagen captada por la cámara de la escena vida sobre la escena del combate, y así aunque se mueva la cámara en el combate, la cámara de la escena vida seguirá captando lo mismo y por tanto las barras de salud no se moverán respecto a la cámara.

4.11. Técnicas especiales

Este apartado explica técnicas de diseño que no son obligatorias en un videojuego, pero que aportan un acabado mucho más profesional. A continuación se explica cómo realizar tres de ellas en *Blender*, y sus posibles aplicaciones a los videojuegos.

4.11.1. Toon shading

Este método sirve para dar a los objetos en *Blender* un aspecto de dibujo animado, rodeando con una línea negra el perfil de un objeto 3d.

Pese a que *Blender* dispone de un botón en la interfaz que permite añadir este efecto, sólo funciona cuando se hace el renderizado de la escena, y no en tiempo real en un juego.

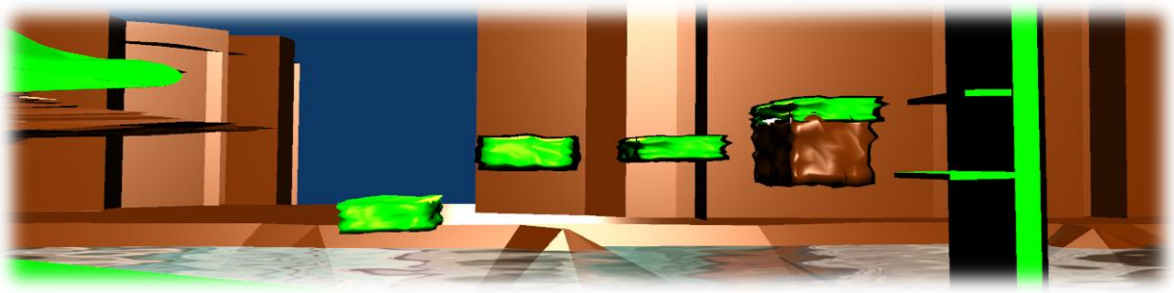


Ilustración 98. Ejemplo de "*toon shading*"

En la ilustración 98 se pueden ver cuatro plataformas verdes, de las que la primera empezando por la izquierda no tiene el perfil negro rodeándola, y las tres siguientes sí que lo tienen. Este perfilado de los objetos no es ni mejor ni peor, simplemente da una visión gráfica distinta a un juego.

Para conseguir el efecto de perfil de dibujos animados, hay que seguir los siguientes pasos:

1. Duplicar el objeto en cuestión y no desplazarlo.
2. Escalar el objeto duplicado, y hacerlo sólo un poco más grande que el objeto original.
3. Entrar en modo edición y seleccionar todos los vértices de la malla del objeto duplicado.
4. Recalcular las normales de las superficies hacia adentro presionando la combinación de teclas "Ctrl + Shift + N"
5. Asignar un material de un color distinto al objeto duplicado, por ejemplo el negro, y asignar la opción "*shadeless*" a dicho material.
6. Colocar 4 fuentes de luz, una en cada esquina del objeto. Si se alejan lo suficiente, las luces pueden servir para los objetos de toda la escena.
7. Por último asegurarse de que está seleccionado el tipo de materiales "GLSL" de *Blender*, ya que con los otros dos tipos no funciona.

Este método se puede aplicar tanto a objetos dinámicos como objetos estáticos, personajes y escenarios, pero siempre es mejor, duplicar los objetos cuando aun no tienen bloques lógicos asignados, ya que estos objetos no deberían reaccionar con el entorno, si no sólo modificar el aspecto visual.

4.11.2. Efectos especiales

Los efectos especiales siempre aportan calidad y espectacularidad a un juego, por eso se incluye este apartado en el manual, en el que se explica una manera muy sencilla y muy utilizada de crear distintos efectos visuales.

Una de las bases de este método, son las imágenes con canales alfa. Los canales alfa permiten que una imagen no tenga fondo, es decir que si de una imagen le eliminamos su fondo, pese a seguir siendo una imagen rectangular, sólo se presentará el dibujo en sí, y no su fondo rectangular. La segunda base, es el actuador que permite añadir objetos dinámicamente a una escena, del que ya se ha hablado en apartados anteriores, y que ahora veremos cómo se aplica más a fondo.

Un efecto especial, puede ser el polvo que levanta una persona al andar, el humo del tubo de escape, efectos luminosos, bolas de energía, la ceniza que cae lentamente de un volcán, además de todo tipo de estelas que puedan dejar los objetos.

Estos efectos podrían hacerse mediante el sistema de partículas de *Blender*, pero como ya hemos dicho antes, no está soportado en el motor de juego y su consumo de recursos es impracticable. También se podría simular humo, por ejemplo creando muchos objetos pequeños con una textura gris semitransparente, pero esto sobrecargaría el procesador y se ralentizaría el juego.

En definitiva, el modo más típico de crear efectos, es simulando pequeños sistemas de partículas con imágenes con canales alfa, de un modo económico en términos de consumo de recursos, y que da muy buenos resultados. Para explicar la técnica, se escogerá un ejemplo en el que una bola levanta polvo del suelo al avanzar por el mismo.

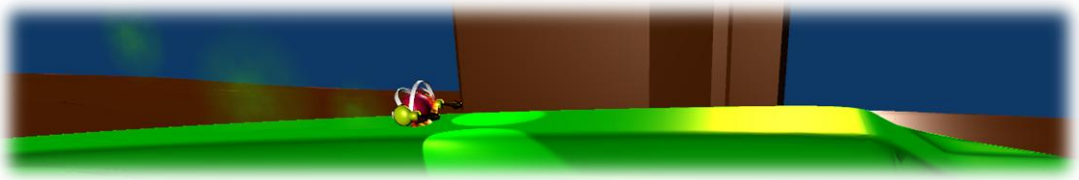


Ilustración 99. Ejemplo de efecto especial de polvo

En la imagen superior se puede ver el polvo levantado por la bola al pasar rodando por la hierba. Ese humo que se ve, no es más que un plano con una textura transparente, creada gracias a una imagen sin fondo. En realidad el efecto está un poco más elaborado, ya que el plano que representa el polvo, se agranda y se eleva lentamente a medida que pasa el tiempo, simulando que el polvo se dispersa y se deja llevar con el viento generado por la esfera.

Para empezar se explicará cómo crear una textura semitransparente con canales alfa:

Para crear este tipo de texturas, lo primero que hay que hacer es acceder a la ventana “UV Image Editor”, cargar la imagen sin fondo en cuestión, seleccionar el objeto sobre el que se quiere colocar la textura, y desenvolverlo como se explica en el apartado “4.2.3.3 Mapeado UV”. Después hay que crear un material nuevo para ese objeto, y crear para ese material una textura nueva de tipo imagen. La imagen seleccionada para crear la textura debe ser la misma que la utilizada en el mapeado UV. De nuevo en el panel de materiales, la textura creada se debe mapear sobre UV, y los parámetros se deben configurar tal y como aparecen rodeados en rojo en la ilustración 100.

Ilustración 100. Configuración de material con textura sin fondo

Una vez se tiene lista esta configuración, se pueden ajustar otros valores como el color, el brillo, la emisión de luz, y en definitiva casi cualquier otro valor distinto de los rodeados en rojo, que no modifique la transparencia del fondo de la textura.

El plano que simula el polvo, o el efecto que sea, debe estar en una capa oculta, y se irá incluyendo en la escena principal cuando se active el sensor dedicado a esa tarea. Por ejemplo el polvo levantado por la bola, sólo se incluye en la escena, cuando la esfera está tocando el suelo, y se mantiene la tecla de avanzar apretada. Respecto al actuador, este añade los planos a intervalos regulares de tiempo, y les aplica una leve fuerza hacia atrás y hacia arriba, como ya se ha explicado antes, para simular la dispersión del polvo.

Para que el plano se mantenga paralelo a la vista de la cámara, y la textura se vea bien, hay que crear una "*IPO*" para el plano que siempre lo mantenga con una rotación determinada. Esa "*IPO*" se ejecutará a partir de un sensor de tipo "Always", para que se mantenga siempre en la misma dirección. Otro modo de arreglar este problema, es utilizar otro tipo de objetos en vez de planos para asignarles las texturas. El problema de esta segunda opción es que los objetos 3D tienen más polígonos, y por tanto consumen más recursos, y que es más difícil mapear una textura plana sobre un objeto 3D que sobre un plano.

Otro ejemplo de aplicación de esta técnica, y que se ha utilizado en el juego de lucha implementado para realizar este manual, es la creación de bolas de energía brillantes. En este ejemplo, al igual que en el anterior, se utiliza un plano para crear las esferas, pero el actuador que las incluye en la escena, lo hace mucho más rápido, y las aplica una fuerza bastante mayor hacia adelante, para que salgan disparadas en dirección a su enemigo.

En este caso en concreto, se desea que las esferas de energía golpeen y reduzcan la salud del enemigo, así que son objetos dinámicos, que cuando chocan contra un luchador, reducen su vida en 1 punto. Puesto que el número de partículas que golpean al enemigo nunca van a ser las mismas, se crea un efecto más realista que permite esquivar parte del ataque, y que crea un ataque más efectivo cuanto más cerca se está del enemigo.

Las ilustraciones 101, 102, 103 y 104 muestran los distintos ejemplos implementados en el juego de lucha.

Ilustración 101. Bolas de energía azules

Ilustración 102. Bolas de energía rojas

Ilustración 103. Puñetazo de energía

Ilustración 104. Efecto de luz al recibir un golpe

4.11.3. Configuración de la cámara

Existen varias maneras de configurar la cámara, dependiendo del juego que se quiera diseñar, y de la vista que se quiera proporcionar en cada momento. En este apartado se explicarán los pasos para configurar algunas de las vistas más típicas de los videojuegos.

Configuración de la cámara para juegos de lucha

Para un juego de Lucha como el que se ha diseñado para escribir este manual, lo normal es tener una cámara, bastante cerca de los luchadores, que a medida que estos se muevan y se alejen el uno del otro, los mantenga dentro de su campo de visión. Para implementar esta capacidad de la cámara, se necesitarán 2 objetos de tipo "*empty*", vacios, que se mantengan siempre en el centro de los dos luchadores. Será necesario crear un controlador en Python para realizar algunos cálculos matemáticos que se explicarán más adelante.

El primero objeto vacío, será al que siga la cámara con la vista, es decir, no desplazará a la cámara, pero sí que hará que rote para que no lo pierda de vista. Esto es útil en ocasiones en las que la lucha de los personajes se desplaza hacia la izquierda o hacia la derecha del ring. En estos casos la cámara girará hacia el lugar en el que los luchadores estén peleando, olvidándose de la parte vacía del ring. Para conseguir esto se utiliza el siguiente script.

```
own = GameLogic.getCurrentController().owner
#se calcula la posicion actual, pa a partirde las posiciones de los luchadores
pa = (GameLogic.pos2[1]+GameLogic.pos1[1])/2
p1 = GameLogic.pos1[1]
p2 = GameLogic.pos2[1]
p = own.position
#se calcula la posicion central entre los luchadores en
#funcion de us posiciones y se guarda en pp
if p1 < 0 and p2 < 0:
    if p1 <= p2:
        pp = (p1 - p2)*-1
    else:
        pp = (p2 - p1)*-1
elif p1 >= 0 and p2 > 0:
    if p2 >= p1:
        pp = p2 - p1
    else:
        pp = p1 - p2
elif p2 >= 0 and p1 < 0:
    pp = p2 - p1
elif p2 < 0 and p1 >= 0:
    pp = p1 - p2
#se modifica la posicion del objeto empty en el eje x
#para que se mantenga siempre en el centro de los luchadores
#la altura y profundidad de laposicion del objeto se mantienen
own.localPosition = [pp*1.5,pa,p[2]]
```

Ilustración 105. Configuración del enfoque de la cámara para un juego de lucha.

Las posiciones pos1 y pos2 son variables globales, que los objetos luchadores actualizan continuamente con su posición. A partir de esas dos posiciones se calcula el centro al que debe mirar la cámara, y se sitúa el objeto vacío en esa posición.

La cámara debe tener un actuador de tipo "*Edit Object*", configurado con el parámetro "*Track To*" y el nombre del objeto vacío al que debe enfocar, activado continuamente por un sensor de tipo "*Always*".

El segundo objeto vacío será padre de la cámara, y se situará en el centro del ring. Con este objeto lo que se hace es alejar o acercar la cámara, en función de la distancia que haya entre los dos luchadores, para que no se salgan de la pantalla a medida que se alejan el uno del otro. El código que aleja y acerca al objeto vacio del ring se muestra a continuación.

```
#se obtiene al objeto dueño del controlador
own = GameLogic.getCurrentController().owner
#se calcula la distancia entre los luchadores
pa = (GameLogic.pos2[1]+GameLogic.pos1[1])/2
p = own.position
#se modifica la profundidad de la posicion del objeto
#en funcion de la disancia de los luchadores
own.position = [p[0],pa,p[2]]
```

Ilustración 106. Código de control de profundidad de la cámara.

Configuración de la cámara para juegos en tercera o primera persona

Para configurar la cámara de juegos en primera o tercera persona, como juegos de acción, o de carreras de coches por ejemplo, la configuración es mucho más sencilla, aunque se pueden aplicar algunos trucos dependiendo del efecto que se pretenda obtener.

Lo primero que hay que hacer es colocar la cámara manualmente detrás y encima del personaje, hasta obtener la vista deseada, y asignarle el personaje como padre. Así la cámara seguirá al personaje siempre en la misma posición con la vista típica de este tipo de juegos. Este método puede plantear un problema, ya que al girar la cámara a la misma velocidad que el objeto al que enfoca, puede dar la impresión de que lo que se mueve no es el coche o la persona en cuestión sino todo el mundo a su alrededor.

Para arreglar este problema, sólo habría que ralentizar el parentesco que existe entre cámara y objeto. Con la cámara seleccionada, habría que ir a la ventana "*Buttons window*", y al icono "*Object*", pulsar el botón "Slow Parent", e introducir el desfase que existirá entre el objeto padre y la cámara, rodeados en rojo en la ilustración 107.

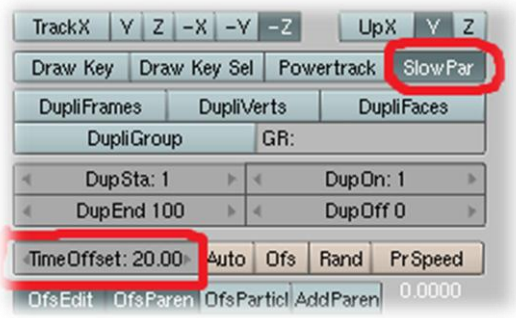


Ilustración 107. *Slow Parent*.

Configuración de la cámara para juegos que mantienen una vista aérea

Típicamente este tipo de juegos suelen tener una vista del personaje en perspectiva desde arriba, y la cámara siempre mantiene la misma orientación. Pese a que la cámara no rote, si que debe seguir al personaje en cuestión, pero si emparentamos la cámara directamente a un objeto, realizará las mismas transformaciones que su padre, incluidas rotaciones. Para evitar que la cámara rote al rotar el personaje, lo que se hace es emparentarla con un punto de la malla que forma parte del objeto, en vez de con el objeto entero, realizando lo que se conoce como "*Vertex Parent*".

Ilustración 108. Ejemplo de cámara aérea perteneciente al juego "*Alien Shooter*"

Para realizar este tipo de parentesco, hay que seleccionar primero la cámara, luego el objeto padre, entrar en modo "*Edit Mode*" pulsando la tecla "Tab", y de la malla del personaje, seleccionar sólo un vértice. Después se pulsa "*Ctrl + P*" y se pulsa en "*Vertex Parent*" en el menú desplegable que aparecerá.

Ya que un punto no tiene volumen, o área, la rotación de un punto es imperceptible, y no tendría sentido almacenar su orientación. Es por esta razón que la cámara no rotará al estar emparentada a un punto.

4.12. Compilación del juego y creación de los ficheros ejecutables

Para crear el ejecutable del juego, hay que pulsar el botón "*File*" de la ventana "*User Preferences*", y del menú desplegable que aparece, seleccionar "*Save Game As Runtime*" como se muestra en la ilustración 109. Se selecciona el directorio donde se quiere guardar el juego y se guarda.

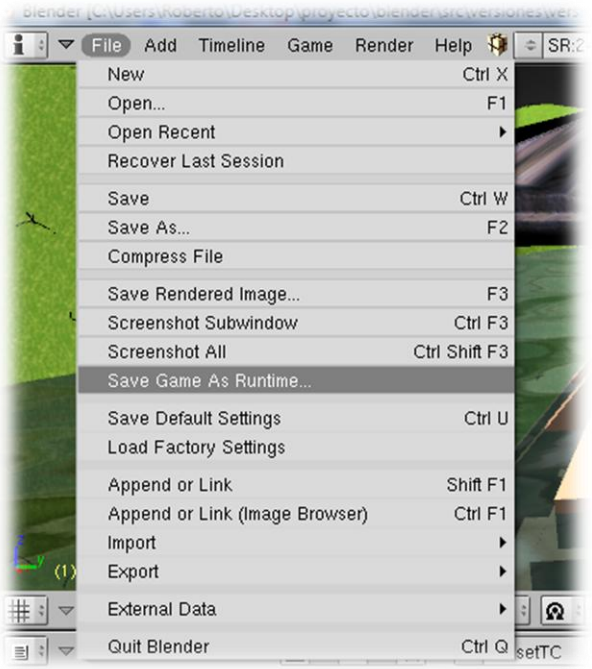


Ilustración 109. Creación del ejecutable del juego.

Esta acción genera un archivo ejecutable, y algunas las librerías necesarias para la ejecución del juego. Hay una librería que hay que descargar y copiar al directorio del juego. Esta librería se llama "vcomp90.dll" y pertenece a Microsoft, pero es un paquete redistribuible y no supone ningún problema legal el incluirlo en el juego. En cualquier caso esta librería sólo es necesaria para generar el juego en Windows.

Para poder ejecutar el juego, será necesario tener instalado "*Python*" [38], que es un software gratuito y multiplataforma.

5. Conclusiones

En un principio, como ya se ha comentado anteriormente, este proyecto pretendía ir enfocado a la creación de un videojuego complejo, pero la ingente cantidad de trabajo que conlleva crear un juego, entre otros motivos, inclinó el foco del desarrollo hacia la realización de un manual que explicase como crear un videojuego con la herramienta *Blender*. De aquí surge la primera conclusión, que es que el diseño e implementación de un videojuego completo y profesional, requiere un equipo de personas cualificadas, y de mucho tiempo de desarrollo y de pruebas.

El segundo punto a tener en cuenta es la cantidad de campos que abraca crear un videojuego, y que sería conveniente disponer de un especialista para desarrollar cada uno de dichos campos. Un diseñador 3D es imprescindible para obtener modelos decentes que den una buena imagen. También sería bueno disponer de alguien experimentado en redes de ordenadores para crear juegos online, una persona dedicada a la animación, no ya por su dificultad si no por el tiempo que lleva crear buenas animaciones, y por ultimo alguien capaz de diseñar la lógica del videojuego y gestionar y coordinar el trabajo de los especialistas nombrados anteriormente.

Las siguientes conclusiones hacen referencia a *Blender*, y a su capacidad para el desarrollo de videojuegos.

Algo que queda patente al comenzar a utilizar *Blender*, es que te anima a seguir adelante, debido a la rapidez con la que se obtienen los primeros resultados, por ejemplo, en pocos minutos se puede tener una esfera que se desplaza por un plano con las flechas del teclado y que puede empujar objetos. Esto es bueno, sobre todo para gente que se está iniciando en el programa o que está explorando diferentes motores de juego para comenzar a desarrollar en el mundo de los videojuegos. Este ánimo inicial se ve apoyado por la gran cantidad de información que existe en internet, que aunque desorganizada, abarca casi todos los ámbitos del desarrollo que ofrece *Blender*. Además *Blender* tiene detrás una gran comunidad siempre dispuesta a resolver dudas y compartir sus conocimientos mediante foros. Por todas estas razones, *Blender* parece una buena opción para empezar a aprender, e introducirse en el desarrollo de videojuegos.

Uno de los puntos fuertes de *Blender*, es que sus módulos, están totalmente integrados, desde el diseño de modelos 3D, hasta la implementación de la lógica, y no sólo eso, si no que dispone de numerosos "*plugins*" con múltiples propósitos, que también pueden sustituir a dichos módulos. Esta integración total, y posibilidad de variación, aporta a *Blender* una abstracción y flexibilidad que permiten a los diseñadores centrarse en implementar el juego en sí, en vez de tener que preocuparse de problemas de compatibilidad de modelos 3D o creación de interfaces con terceros programas.

Pese a que los bloques lógicos son un sistema original y bastante rápido para implementar la lógica de los juegos, puede ser insuficiente para objetos que tengan una lógica muy complicada, no porque no se pueda llevar a cabo, si no porque los diagramas de bloques pueden quedar demasiado complicados, y su depuración puede convertirse en una difícil tarea.

Una de las desventajas de *Blender* es que no permite, o no dispone de librerías que soporten la creación o importación de objetos que no existan ya en el archivo de juego en tiempo de ejecución. Es decir que no se puede crear un personaje en un archivo, y decir de algún modo al archivo principal que lo incluya cuando ocurra algún evento. Para hacer esto, habría que importar al personaje en tiempo de desarrollo, y colocarlo en una capa oculta para añadirlo dinámicamente mediante la activación de algún sensor. Esto puede suponer un problema a la hora de hacer el juego extensible, ya que si se quieren añadir nuevos personajes por ejemplo, habría que rehacer y recompilar código, y el espacio donde almacenar todos los objetos estaría limitado.

Algo que lleva a confusión muchas veces a la hora de utilizar *Blender*, es que muchas de las funcionalidades disponibles para la creación de videos y animaciones no funcionan en tiempo real, en el motor de juego, pero aun así se permite que se utilicen. Esto puede llevar a confusión y a una gran pérdida de tiempo si no se conocen en detalle esas limitaciones, ya que se puede pensar que se está haciendo algo bien, y luego no funcionar sin saber porqué.

Viendo varias de las desventajas expuestas, se podría decir que *Blender* no es una plataforma propicia para desarrollar un juego profesional y complejo, pero también es cierto que es un proyecto que pese a ser software libre y de código abierto, tiene un equipo de desarrollo detrás, que lo mejora día a día, y cuyo trabajo se ve respaldado por cientos de usuarios que ayudan a depurarlo y aportan nuevas ideas al mismo continuamente.

Como conclusiones personales referentes al proyecto en sí mismo, se han conseguido los objetivos perseguidos, desde la creación del manual, hasta la reducción del esfuerzo debido al desconocimiento de la herramienta de diseño. Posteriormente a la realización del manual se ha implementado un videojuego muy sencillo en menos de la mitad de tiempo de lo que se habría tardado sin tener estos conocimientos, poniendo en práctica varios de los métodos aprendidos. De este hecho se deduce que los conocimientos respecto a *Blender* se han asimilado bien.

Personalmente, ha sido muy satisfactorio poder realizar y centrarme un trabajo tan amplio con éxito, y sobre todo el poder aplicar la informática, a uno de los campos que siempre me han gustado, como es el de los videojuegos.

6. Líneas futuras

Este manual da lugar a muchas posibilidades futuras. La aplicación más directa, sería la de crear videojuegos basándose en él. También cabe la posibilidad de extender este manual, explicando conceptos más concretos, exponiendo más ejemplos, y aportando nuevas ideas que no figuren en este escrito.

Pese a que se ha dejado a un lado la parte de animación y creación de películas en *Blender*, muchos de los conocimientos son compartidos con los del motor de juego, y se podría crear un manual sobre cómo crear películas con esta herramienta, basándose en este proyecto.

Con los conocimientos aprendidos tras la lectura de este manual, no sólo se pueden crear juegos, si no que se pueden aplicar dichos conocimientos a otros campos de la industria. Por ejemplo se podría utilizar para generar simulaciones físicas, gracias a su motor de físicas. Con su herramienta de diseño 3D y su capacidad para animar objetos, es posible crear interfaces llamativas, que se podrían integrar fácilmente con cualquier programa utilizando Python.

Otro campo en el que la visualización es importante, es en la monitorización. Por ejemplo se puede utilizar para monitorizar sistemas informáticos en paralelo, mostrando la carga de los distintos procesadores con una visión de los mismos en 3D, mostrando si alguno de los equipos se ha caído, o cuáles de ellos están ociosos.

Con *Blender* también sería posible crear programas educativos. Para gente con conocimientos avanzados de informática, puede ser un modo de entrar en una disciplina de la Informática que no se toca demasiado en las universidades. Por ejemplo se podría introducir la herramienta *Blender* en alguna clase como “modelado de entornos virtuales” o “multimedia” que tratan estos temas pero con programas bastante más arcaicos y anticuados. Y para gente más joven, puede ser un modo de mostrarles la parte más visual y llamativa de la informática, y acrecentar su interés en la misma.

7. Gestión del proyecto

En este apartado se expone la planificación inicial que se hizo para tener una idea del tiempo que iba a llevar la ejecución de las distintas tareas del proyecto. Para ello se ha utilizado un diagrama de Gantt que se comentará más adelante en este mismo apartado.

Además de la planificación inicial, se ha creado otro diagrama de Gantt con todas las tareas que conforman el desarrollo del proyecto, y con los tiempos que se han dedicado realmente a cada una de ellas. Este segundo diagrama se utilizará para compararlo con el primero, y sacar unas conclusiones sobre los factores que han influido en los cambios sobre la planificación inicial.

Como se podrá apreciar en las siguientes imágenes, la diferencia entre los dos diagramas, se debe principalmente al cambio de objetivo que ocurrió durante el desarrollo del proyecto, y que cambió de ser la implementación de un juego, a la creación de un manual sobre cómo hacer videojuegos en *Blender*. Además de este motivo que ya se comentó en la introducción, hay diversos motivos que han extendido la duración del proyecto, y que se comentarán a continuación.

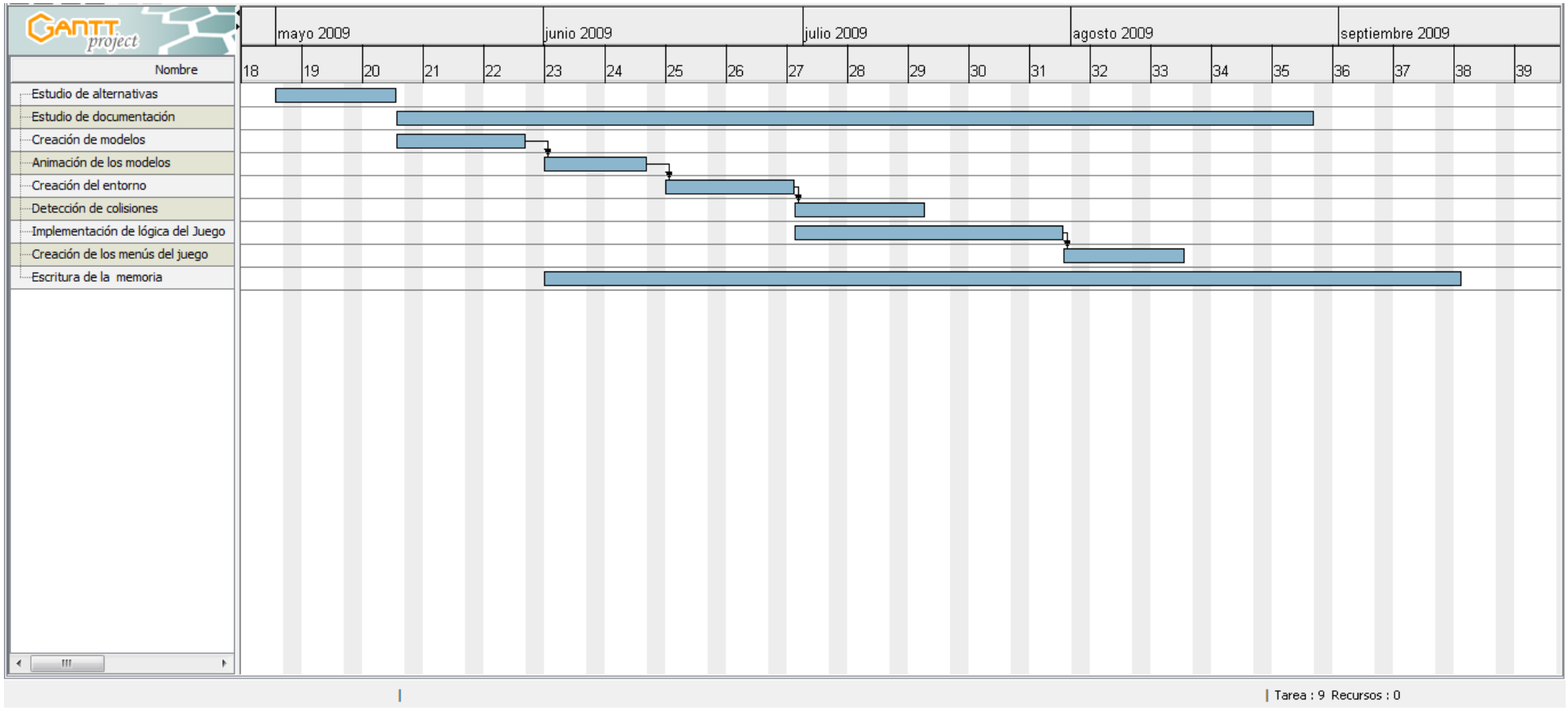


Ilustración 110. Diagrama de Gantt inicial

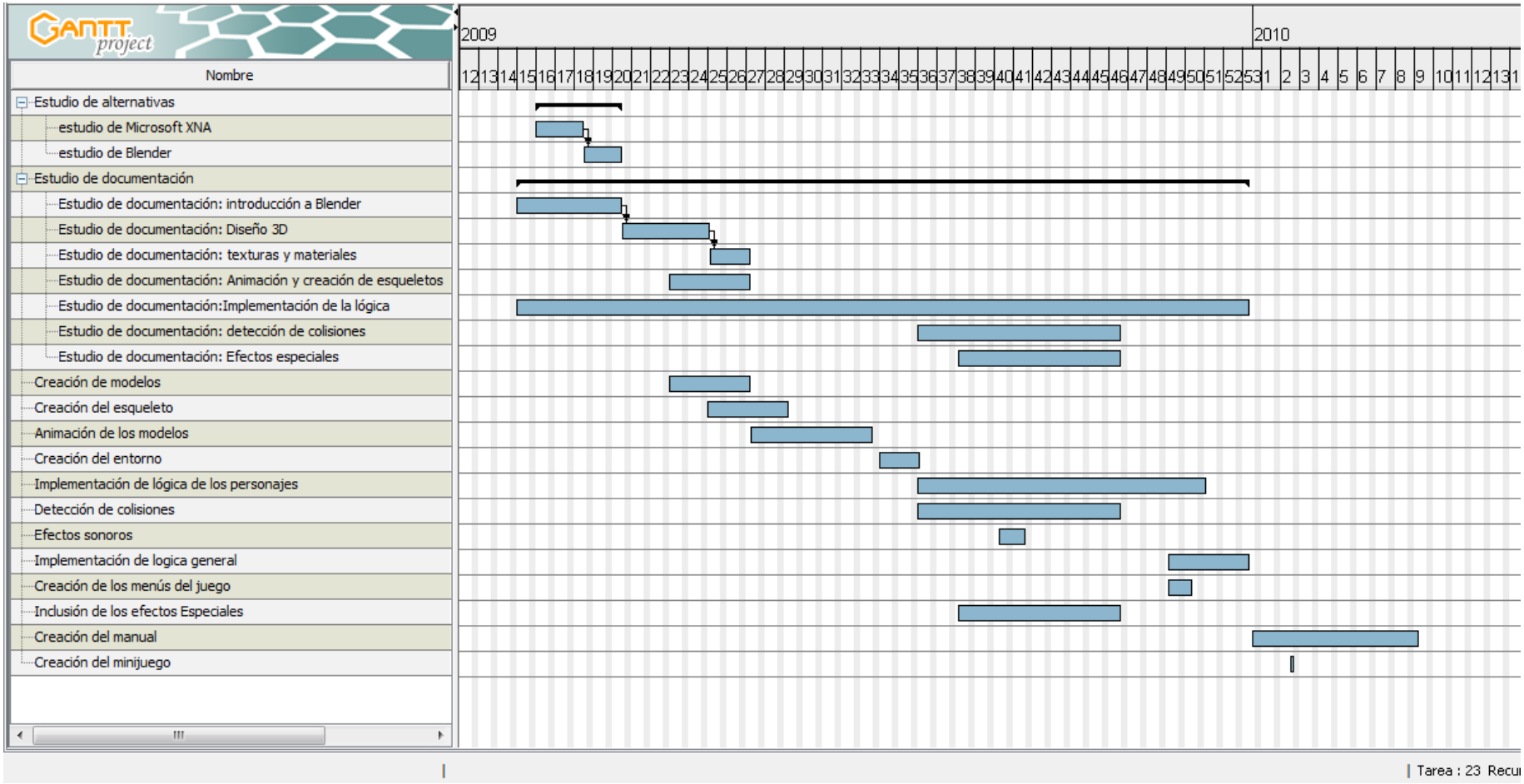


Ilustración 111. Diagrama de Gantt real

Además de los dos diagramas anteriores, se utilizará un gráfico de barras comparativo, que muestra las diferencias de tiempo entre el tiempo planificado y el real.

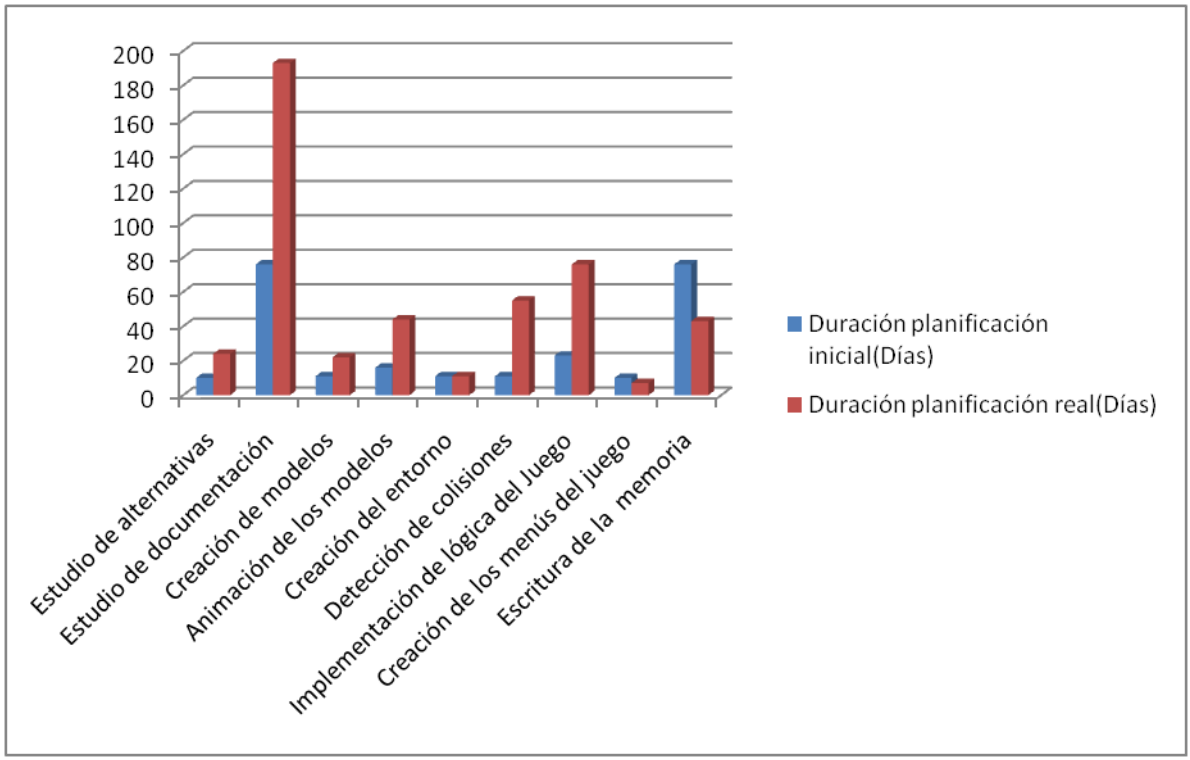


Ilustración 112. Gráfico comparativo de tiempos de planificación

Como se puede observar en la ilustración 112, prácticamente la duración real de todos los tiempos supera a la duración planificada. Esta diferencia se debe en gran medida al desconocimiento que se tenía sobre el campo del desarrollo de videojuegos, pero se va a comentar tarea a tarea la diferencia de tiempo, para conocer mejor las causas de esta varianza.

Respecto al estudio de alternativas no se calculó bien la profundidad a la que habría que llegar para poder tomar una decisión, de ahí que el tiempo de estudio se doblase.

El hecho de que la duración del estudio de la documentación difiera tanto, es porque en ambos casos, el estudio de la documentación se realiza a lo largo del desarrollo de todo el proyecto, y puesto que el resto de tareas han alargado la duración real del mismo, en consecuencia se ha incrementado proporcionalmente el tiempo de estudio de la documentación.

La creación de modelos también llevó bastante más tiempo del esperado, diseñar un personaje en 3D es muy difícil, y requiere ciertas dotes artísticas, así que después de intentar crear un luchador realista, y un stickman, y ver en la etapa de animación que era imposible que las animaciones quedasen realistas con estos modelos, se opto por descargar unos modelos de luchadores prefabricados “open source” y “royalty free”.

Aunque parezca que el diseño de personajes no tuvo utilidad alguna, no fue así. Los conocimientos de diseño 3D adquiridos en esa etapa agilizaron mucho la tarea de creación de escenarios, cuya planificación inicial coincide con la planificación real obtenida.

La diferencia de tiempo entre la estimación, y la duración real de la tarea de detección de colisiones, es enorme. Lo mismo ocurre con la tarea de implementación de lógica. Este hecho es debido a que en la planificación inicial, se pretendía llevar a cabo una tarea primero y la otra después, pero cuando se estaba implementando el código, se vio que era necesario implementar la lógica al mismo tiempo que las colisiones para poder probar la efectividad de las mismas.

Una vez asimilados los conceptos de diseño 3D y de implementación de lógica, crear los menús del juego fueron una tarea sencilla, y esa es la razón de que el tiempo planificado sea superior al tiempo real dedicado a la tarea.

Por último, el tiempo planificado para la escritura de la memoria también ha sido superior a la duración real. Esto es debido al cambio de objetivos del proyecto mencionado anteriormente. En un principio se pretendía crear un juego, y en la memoria explicar cómo se había creado dicho juego. En esta aproximación inicial, se pretendía hacer la memoria a medida que se iba implementando el programa, y por lo tanto la duración de su escritura duraría casi tanto como la totalidad del proyecto.

Al final se decidió escribir un manual para crear videojuegos en *Blender*, y debido al formato y estructura que debe guardar un manual, no se podía empezar a escribir sin tener una visión global de cómo funcionaba *Blender*, y el detalle al que se iba a llegar. En este caso la memoria adquiere una mayor relevancia que el programa en sí, y aunque según el gráfico comparativo se le dedica menos tiempo que el tiempo planificado, si nos fijamos en los diagramas de Gantt, veremos que el tiempo planificado de la memoria va en paralelo al resto de tareas, y el tiempo real, es tiempo añadido al final del proyecto, y dedicado exclusivamente a la memoria.

8. Bibliografía

Libros utilizados

Carsten Wartmann, M. K. *the Blender Game Kit 2nd Edition.* Amseterdam, Holanda: Blender Foundation.

Historia de los motores de juego

http://www.maximumpc.com/article/features/3d_game_engines?page=0,0

http://www.naturalmotion.com/euphoria.htm

Características de Blender

http://www.*Blender*.org/

http://www.*Blender*.org/features-gallery/features/

http://www.*Blender*.org/development/

http://www.*Blender*.org/community/user-community/

YoFrankie

http://www.yofrankie.org/

Información sobre motores de juego

http://www.devmaster.net/engines/list.php?

http://www.neoteo.com/top-10-los-motores-graficos-mas-importantes.neo

http://www.develop-online.net/news/32250/The-top-10-game-engines-revealed

http://pc.ign.com/articles/100/1003725p1.html

Video tutoriales seguidos:

http://www.youtube.com/watch?v=9cTnPFPxyCs

http://www.youtube.com/watch?v=6_NfNjGTp_0

9. Referencias

[1] http://www.microsoft.com/games/
[2] http://www.ubi.com/ES/default.aspx
[3] http://www.electronicarts.es/
[4] http://www.blender.org/
[5] http:// www.ogre3d.org
[6] http:// www.idsoftware.com
[7] http://en.wikipedia.org/wiki/Build_engine
[8] http://www.3drealms.com/duke3d/index.html
[9] http://www.bethsoft.com/spa/index.php
[10] http://es.wikipedia.org/wiki/Z-Buffer
[11] http:// www.opengl.org/
[12] http://en.wikipedia.org/wiki/RenderWare
[13] http://www.idsoftware.com/games/quake/quake/
[14] http://half-life.wikia.com/wiki/GoldSrc
[15] http://www.unreal.com/
[16] http://en.wikipedia.org/wiki/Geo-Mod
[17] http://www.emergent.net/Products/Gamebryo/Gamebryo-26/
[18] http://www.naturalmotion.com/euphoria.htm
[19] http://www.devmaster.net/engines/list.php?
[20] http://www.develop-online.net
[21] http://pc.ign.com
[22] http://www.neoteo.com
[23] http://www.crytek.com/technology/cryengine-3/specifications/
[24] http://www.**unreal**technology.com/
[25] http://www.1up.com/do/previewPage?cId=3167534&p=37
[26] http://www.povray.org/
[27] http://vray.info/
[28] http://www.luxrender.net/
[29] http://blogs.gamefilia.com/i-yova-i/16-02-2009/19371/tecnicas-cel-shading
[30] http://en.wikipedia.org/wiki/Shading
[31] http://bulletphysics.org/wordpress/
[32] http://www.siggraph.org/
[33] http://projects.Blender.org
[34] http://www.Blender.org/development/report-a-bug/
[35] http://www.yofrankie.org/
[36] http://e2-productions.com/repository/index.php
[37] http://wiki.blender.org/index.php/Doc:ES/Manual/Textures/UV/Unwrapping_a_Mesh
[38] http://www.python.org/